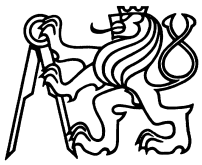
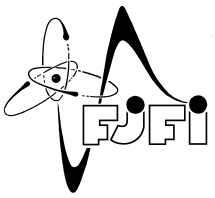


Czech Technical University in Prague
Faculty of Nuclear Sciences and Physical Engineering



Automated Translation of Ancient Texts into Modern Language

Automatizovaný překlad starověkých textů do současného jazyka

Research Project

Author: **Bc. Katka Morovicsová**

Supervisor: **Ing. Radek Mařík, CSc**

Academic year: 2025/2026

ZADÁNÍ VÝZKUMNÉHO ÚKOLU

I. OSOBNÍ A STUDIJNÍ ÚDAJE

Jméno a příjmení: Bc. Katka Morovicsová

Fakulta/ústav: **Fakulta jaderná a fyzikálně inženýrská**

Zadávající katedra/ústav: **Katedra matematiky**

Studijní program: **Aplikované matematicko-stochastické metody**

Specializace:

II. ÚDAJE K VÝZKUMNÉMU ÚKOLU:

Název výzkumného úkolu:

Automatizovaný překlad starověkých textů do současného jazyka

Název výzkumného úkolu anglicky:

Automated Translation of Ancient Texts into Modern Language

Pokyny pro vypracování:

Pro překládání starověkých textů s tzv. omezenými zdroji nelze plně využít současné překladače vyžadující velké objemy textů ve fázi trénování. Manuální překládání zahrnuje i velmi náročné aktivity, např. vyhledávání podobných textů. Cílem úkolu je sestavení procesu překladu se zaměřením na egyptské hieroglyfy. Doporučený postup:
1. Vyhledejte metody vhodné pro překlad staroegyptských textů do transliterace a dále po detekci lingvistických struktur pro překlad do moderního jazyka, např. angličtiny. Výběr zohlední problém jazyka s omezenými zdroji. Vytvořte přehled metod a jejich vlastností.
2. Sestavte vhodný procesní řetězec, jehož optimalizace zahrne i odhadování hyper-parametrů modelů.
3. Experimentálně ověřte přesnost zpracování a proveďte diskusi dosažených výsledků.

Seznam doporučené literatury:

[1] H. Lane, C. Howard, H. M.Hapke, Natural Language Processing in Action, Understanding, Analyzing, and Generating Text with Python, Manning Publications Co., New York, 2019
[2] T. Ganegedara, Natural Language Processing with Tensorflow, Packt Publishing, Birmingham, 2018
[3] L. Deng, Y. Liu, Deep Learning in Natural Language Processing, Springer, New York, 2018
[4] S. Rosmorduc. Automated Transliteration of Late Egyptian Using Neural Networks. Lingua Aegyptia - Journal of Egyptian Language Studies, 28:233–257, December 2020.

Jméno a pracoviště vedoucí(ho) výzkumného úkolu:

Ing. Radek Mařík, CSc. FEL-ČVUT

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) výzkumného úkolu:

Datum zadání výzkumného úkolu: **31.10.2024** Termín odevzdání výzkumného úkolu: **09.05.2025**

Platnost zadání výzkumného úkolu: **2 roky**

podpis vedoucí(ho) práce	podpis garanta stud. programu	podpis vedoucí(ho) katedry

III. PŘEVZETÍ ZADÁNÍ

Student bere na vědomí, že je povinen vypracovat výzkumný úkol samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací. Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést ve výzkumném úkolu.

5. 11. 2024
Datum převzetí zadání

Podpis studenta

Acknowledgment:
I would like to thank Ing. Radek Mařík, CSc. for his expert guidance and supervision throughout this work.

Author's declaration:
I declare that this Research Project is entirely my own work. All experiments, analyses, and conclusions presented in this work were conducted and developed by myself. Language models such as ChatGPT and Claude were used for language correction and proofreading purposes. I am fully responsible for all contents, results, and conclusions presented in this work, and I have listed all the used sources in the bibliography.

Prague, Bc. Katka Morovicsová

Název práce:

Automatizovaný překlad starověkých textů do současného jazyka

Autor: Bc. Katka Morovicsová

Studijní program: Aplikované matematicko-stochastické metody

Specializace:

Druh práce: Výzkumný úkol

Vedoucí práce: Ing. Radek Mařík, CSc, České vysoké učení technické v Praze, Fakulta elektrotechnická, Katedra telekomunikační techniky

Abstrakt: Tato práce se zabývá automatizovaným překladem starověkých textů, konkrétně transliterací staroegyptských hieroglyfů. Zaměřuje se na systematické srovnání čtyř metod optimalizace hyperparametrů – Grid Search, Random Search, Tree-structured Parzen Estimator (TPE) a Gaussian Process (GP) – pro trénování transformerových modelů na překladu jazyků s omezenými zdroji. Výsledky experimentů ukazují, že bayesovské optimalizační metody (TPE a GP) dosahují lepších výsledků než nebayesovské přístupy. Nejlepší model nalezený pomocí automatické optimalizace dosahuje výsledků srovnatelných s ručně laděnými modely, přičemž jeho nalezení vyžaduje výrazně méně času a úsilí. Dále tato práce zkoumá vztah mezi přesností modelu a velikostí použitého trénovacího datasetu a identifikuje plató, které ukazuje, jaké minimální množství dat je potřeba k natrénování transformeru na hieroglyfických datech.

Klíčová slova: bayesovská optimalizace, jazyky s malým množstvím zdrojů, optimalizace hyperparametrů, starověká egyptština, strojové učení, transformerové modely, transliterace

Title:

Automated Translation of Ancient Texts into Modern Language

Author: Bc. Katka Morovicsová

Abstract: This work addresses automated translation of ancient texts, specifically the transliteration of Ancient Egyptian hieroglyphs. It focuses on a systematic comparison of four hyperparameter optimization methods-Grid Search, Random Search, Tree-structured Parzen Estimator (TPE) and Gaussian Process (GP)-for training transformer models for low-resource language translation. Experimental results show that Bayesian optimization methods (TPE and GP) achieve better performance than non-Bayesian approaches. The best model found through automated optimization reaches performance comparable to manually tuned models, while requiring significantly less time and effort. Furthermore, this work investigates the relationship between model performance on training dataset size and identifies a performance plateau, indicating the minimum amount of data required to successfully train transformer models on hieroglyphic data.

Key words: Ancient Egyptian, Bayesian optimization, hyperparameter optimization, low-resource languages, machine learning, transformer models, transliteration

Contents

Chapter 1

Introduction

Machine learning methods for language processing typically rely on large volumes of annotated data, a requirement that is a significant challenge for low-resource languages. This work focuses on the translation of Ancient Egyptian, specifically on one part of the translation process - transliteration.

Prior work on automatic transliteration of Ancient Egyptian has primarily relied on recurrent neural network architectures, with Rosmorduc's model [Rosmorduc 2020] standing as the state-of-the-art benchmark in the field. In contrast, transformer-based experiments conducted in my Bachelor's thesis [Morovicsová 2024] demonstrated that a carefully designed transformer model can match this performance.

Hyperparameters play a central role in determining the behavior and performance of machine learning models. In contrast with parameters learned during training, hyperparameters control structural and training-related aspects of a model—such as the number of layers, hidden dimensions, and attention configurations in transformer architectures. These choices define the model's capacity to capture patterns, its ability to generalize from limited data, and its overall training dynamics. An unsuitable configuration of hyperparameters can lead to underfitting, overfitting, unstable training, or poor convergence.

Because of their importance, selecting appropriate hyperparameters is often a time-consuming part of model development. Early systematic approaches such as random search [Bergstra and Bengio 2012] showed that simple methods can outperform exhaustive grid search while being more efficient in high-dimensional spaces. More recent work [Snoek, Larochelle, and Adams 2012] demonstrated that Bayesian optimization can further improve the efficiency of hyperparameter search by using information from previous trials to guide the exploration of the search space. Several modern frameworks build on these ideas to automate hyperparameter tuning. In this work, automatic hyperparameter optimization is used to reduce manual trial-and-error and to efficiently identify good model configurations.

Although previous findings [Morovicsová 2024] confirm the potential of a transformer model for low-resource translation tasks, achieving good performance depends heavily on selecting appropriate hyperparameters. The model must generalize from extremely limited data, meaning that even small changes in hyperparameters can lead to substantial differences in model behavior and stability. This sensitivity underscores the need for a structured exploration of the hyperparameter space rather than relying on a trial-and-error approach.

To investigate this systematically, this work evaluates four optimization strategies: Tree-Parzen Estimators, Gaussian Processes, Random Search, and Grid Search to determine which method is best suited for identifying good hyperparameter configurations. This work addresses two central research questions:

1. Which hyperparameter optimization method is the most effective for the task of Ancient Egyptian transliteration?

2. Can a systematic optimization framework discover a model that improves upon the transformer developed through manual tuning?

The results indicate that Bayesian optimization methods consistently identify the strongest models. Furthermore, the results show that systematic hyperparameter optimization can identify transformer configurations that perform comparably to carefully manually tuned models. The best configuration found through automated optimization achieves similar results to the previously developed model while slightly improving on several metrics such as Levenshtein distance, ROUGE-L, and SacreBLEU. Importantly, these results required substantially less manual effort: while manual tuning needed several weeks of trial-and-error experimentation, the automated search identifies competitive configurations within a significantly shorter time frame. These findings suggest that automatic hyperparameter optimization is an effective and practical alternative to manual tuning for low-resource transliteration tasks.

Furthermore, this work also looks at several related factors that influence model performance beyond hyperparameter optimization. One important aspect is the size of the available training data. Determining the minimum amount of data required to successfully train a neural model is crucial, particularly in low-resource settings. Unlike high-resource language tasks, where additional data can often be collected or generated, Ancient Egyptian transliteration relies on manually curated sources, typically produced by expert Egyptologists. As a result, the amount of training data is severely limited and cannot be easily expanded.

This creates a clear gap between the amount of data that would ideally be required for neural training and the data that is available realistically. Understanding this gap is important for evaluating whether neural approaches are appropriate for the task or whether alternative methods, such as rule-based or hybrid systems, might be better under data constraints. In addition to data size, this work examines the effect of overall model size, measured by the number of trainable parameters, as well as the sensitivity of the models to random initialization. Together, these analyses provide a broader view of the factors that affect model stability, performance, and practical applicability in low-resource transliteration tasks.

The following work is divided into several sections. Chapter 2 first goes through problems of machine translation of low-resource languages and then continues with explanation of different hyperparameter tuning methods and their advantages. Chapter 3 outlines the design of different experiments conducted as part of this work, and Chapter 4 further reports and analyzes the corresponding results.

Chapter 2

Related Works

This chapter reviews prior works that motivate and contextualize this work. First, it surveys machine learning research relevant to low-resource languages, with a focus on the specific challenges posed by Ancient Egyptian. This chapter further introduces hyperparameter optimization as a key methodological component of modern machine learning, outlining common strategies from grid and random search to Bayesian optimization with Gaussian processes and the Tree-structured Parzen Estimator, which are relevant for training reliable models under limited and noisy data conditions.

2.1 Machine Learning and Low-Resource Languages

Translating ancient scripts like hieroglyphs using machine learning is a difficult task. This is because hieroglyphs combine the challenges of a complex writing system with the problems of working with a low-resource language. This section explains these difficulties and why automatic translation of hieroglyphs is especially hard.

Ancient Egyptian is a language which can be called **low-resource** [Haddow et al. 2022], meaning there is only a small amount of parallel data (sentences paired with their translations) available. For common languages like English and French, millions of such pairs exist, however, for low-resource languages, there are only a few thousand or even less.

Low-resource languages often have noisy or incomplete data. They usually lack large dictionaries, monolingual corpora, or tools that help with language analysis. Because of this, standard machine translation methods that work well on popular languages do not perform as well on low-resource languages [Haddow et al. 2022].

2.1.1 Rule-Based approaches for low-resource languages

Rule-based systems, which rely on predefined linguistic rules and dictionaries rather than large training corpora, have been applied to hieroglyphic translation tasks [Gardiner 1957]. For example, the Hieroglyphic E-Convert Application (HECA) uses a dictionary-based approach derived from Gardiner's Egyptian Grammar to convert transliterated text into hieroglyphs [Washington 2003; Gardiner 1957].

2.1.2 Machine Learning for low-resource languages

Previous works include: training a neural network model for automated transliteration of Late Egyptian, showing that deep learning can learn consistent mappings from textual input to transliteration even in a limited-data setting [Rosmorduc 2020], transliteration models using word-level transliteration for

Egyptian words using data from the Ramses Transliteration Corpus [H. Jauhiainen and T. Jauhiainen 2023] and more recently, a Hieroglyphic Transformer for translating Ancient Egyptian into modern languages, adapting a multilingual Transformer (based on M2M-100) and training it on Thesaurus Linguae Aegyptiae (TLA) dataset [De Cao et al. 2024].

2.1.3 Translation of Hieroglyphs

The task of translating hieroglyphs is especially difficult because:

- There is very little parallel data available, since many texts are damaged, incomplete, or not digitized.
- The writing system is complex, with over 1,000 different glyphs, and symbols that can mean different things depending on context [Ardagh 1999].
- The domain is specialized, with religious, administrative, or funerary texts, which do not have large modern corpora for comparison.
- There are very few supporting tools or large dictionaries for hieroglyphic scripts, as most of the translation is done manually by experts.

Furthermore, Ancient Egyptian changed significantly over more than three millennia (from roughly 3200 BCE to 395 CE [Shaw 2003]), which means that the grammar changed as well. This makes using Ancient Egyptian data particularly challenging.

Briefly, the translation of Ancient Egyptian has the following steps:

1. Capturing source data (e.g. taking photographs of a tomb wall)
2. Redrawing of the photographs by a specialist
3. Identifying the specific hieroglyphs (often challenging as there is data missing)
4. Transliteration, see Section 2.1.3.1
5. Translation into another language

For more information about the challenges of automated translation of hieroglyphs, see [Morovicsová 2024].

2.1.3.1 Transliteration

Transliteration of Ancient Egyptian is the process of converting the original hieroglyphic symbols into a standardized alphabetic or symbolic script that represents the phonetic components of the language. This intermediate step is important in the translation pipeline, as it transforms complex glyphs into a more accessible format suitable for analysis and further linguistic processing.

For more information about transliteration, see [Rosmorduc 2020; Morovicsová 2024].

2.1.4 Data

The dataset used in this work was shared by Serge Rosmorduc on GitLab in 2020. It contains sentences encoded in **Gardiner codes** (hieroglyphs) and their corresponding **Manuel de Codage**[1] transliterations. It does not include English or French translations [Rosmorduc 2020].

The data originates from the **Ramses corpus**[2], which at the time contained approximately 500,000 words. The dataset includes both publicly available texts and some that had not yet been proof-checked. Missing transliterations were generated using automatic database lookup, which may introduce errors. For a concise overview of the corpus information, see Table 2.1.

Characteristic	Value / Description
Source language periods	Mostly Late Egyptian, some Middle Egyptian
Script distribution	7% hieroglyphic, 93% hieratic
Total corpus size	~500,000 words
Training set size	66,693 hieroglyph–transliteration pairs
Validation set size	1,841 pairs
Test set size	2,729 pairs
Data source	Ramses Corpus[2] via Rosmorduc (2020)
Encoding	Gardiner codes (source), MdC transliteration (target)

Table 2.1: Summary of the Ramses dataset composition

Each corpus subset (training, validation, and testing) contains three text files:

- Source sentences in Gardiner codes (no word spacing)
- The same sentences with word separation
- Corresponding transliterations in MdC format

These files are ordered correspondingly, so each line in the source dataset has a matching transliteration on the same line in the target dataset.

2.1.4.1 Example of Source and Target Pair

To provide an example of the data available (similarly to [Morovicsová 2024]) see below an example of a source sentence (in Gardiner codes with word spacing) and its corresponding transliteration (in MdC):

```
M17 Z7 _ M17 Z7 _ A1 _ D21 _ S29 G36 D21 N35A A2 _ M17 G17 _ I9 _
A47 G1 Z7 Z4 Y1 A24 _ X1 Z7 _ D21 Z3A _ D58 U30 G1 Z7 K2 G37 X1 Z7 _ I9 Z3A _

i w _ i w _ = i _ r _ s w r _ m _ = f _
s A w _ t w _ r _ b w . t _ = f _
```

Each symbol or letter is separated by a space, and words are separated by underscores.

[1] Manuel de Codage is a standardized system for the digital encoding of Egyptian transliteration. It was developed in the 1980s by the International Association of Egyptologists. It is a set of rules for representing hieroglyphs using ASCII characters.

[2] `http://ramses.ulg.ac.be/site/aboutRamses`

2.1.5 Data-availability in ancient Egyptian corpora

To give more context to the amount of data available, here is an overview of available ancient Egyptian datasets commonly used for language and translation tasks, along with their approximate sizes and historical periods.

- **Pyramid Texts** (Old Kingdom, approx. 2686–2181 BCE): A small but foundational corpus of royal funerary inscriptions containing around 700–760 spells [Allen 2005].
- **Coffin Texts** (Middle Kingdom, approx. 2055–1650 BCE): A more extensive funerary-text corpus with roughly 1,100–1,200 spells preserved in the main editions.[3]
- **Thesaurus Linguae Aegyptiae (TLA)**: The principal digital corpus containing annotated texts from multiple Egyptian periods, widely used for linguistic and translation research. It includes several major subcorpora:
  - **Earlier Egyptian** (Old and Middle Egyptian): approximately 12,773 fully intact, lemmatized sentences in the curated subset suitable for machine learning.[4]
  - **Late Egyptian** (New Kingdom and later): around 9,005 sentences, as reported in recent research datasets.[5]
  - **Demotic** (Late Period to Roman Egypt): lemma-token counts exceed 300,000, making it the largest component of the TLA corpus.[6]
- **Ramses Corpus** (see Section 2.1.4)

Compared to these datasets, the Ramses corpus is relatively large, covering roughly 67,000 sentences.

2.2 Hyperparameter Tuning in Machine Learning

Hyperparameter tuning is a critical process in building machine learning (ML) models. Hyperparameters — such as number of layers, hidden dimensions, and attention configurations influence how the learning algorithm behaves. Their initial setting can drastically affect the outcome of the training. Finding good settings is challenging because the relationship between hyperparameters and model performance is typically nonlinear and noisy, and evaluation is expensive: each trial often involves full model training and validation [Snoek, Larochelle, and Adams 2012].

This problem is related to active learning, which focuses on selecting informative queries to improve learning when evaluations are expensive [Cohn, Atlas, and Ladner 1994]. While both active learning and hyperparameter optimization use uncertainty-aware, sequential decision making to limit costly evaluations, active learning is not further explored in this work.

This section reviews key optimization strategies that have been proposed to automate and improve the process of hyperparameter optimization.

[3] Database of the Coffin Texts spells. https://www.britannica.com/topic/Coffin-Texts

[4] https://huggingface.co/datasets/thesaurus-linguae-aegyptiae/tla-Earlier_Egyptian_original-v18-premium

[5] https://aclanthology.org/2025.alp-1.12.pdf

[6] https://thesaurus-linguae-aegyptiae.de/info/text-corpus

2.2.1 Definitions

The following definitions and notation are primarily adapted from the standard literature on hyperparameter optimization, including the comprehensive tutorials by Brochu et al. [Brochu, Cora, and Freitas 2010] and Snoek et al. [Snoek, Larochelle, and Adams 2012].

The **hyperparameter space** is denoted by $\mathcal{X} = X_1 \times X_2 \times \cdots \times X_d$, where each X_i represents the set of possible values for the i^{th} hyperparameter, and d is the total number of hyperparameters being optimized.

An **objective function** $f(x)$ is a mapping from a set of hyperparameters x to a numerical score that measures how well the model performs with those hyperparameters:

$$f : \mathcal{X} \to \mathbb{R}, \quad f(x) = \text{validation\_loss(model trained with } x).$$

The goal of hyperparameter tuning is to find the **optimal configuration** of hyperparameters x^* that minimizes (or maximizes) this function:

$$x^* = \arg\min_{x \in \mathcal{X}} f(x).$$

In practice, evaluating $f(x)$ can be expensive because it often requires training and validating a full model.

An **epoch** refers to one complete pass of the training algorithm over the entire training dataset. During each epoch, the model parameters are updated based on the training data.

A **trial** denotes a single evaluation of the objective function $f(x)$ for a specific hyperparameter configuration x. Each trial involves training the model with the chosen hyperparameters, usually for a fixed number of epochs, and then computing the validation performance. Hyperparameter optimization consists of performing multiple trials.

2.2.2 Grid Search

Grid Search [Bergstra and Bengio 2012; Hsu, Chang, and C.-J. Lin 2003] is the simplest approach: it defines a grid over the hyperparameter space and evaluates the model for every possible combination of values, meaning the grid search evaluates all points $x \in \mathcal{X}$.

While conceptually simple and easy to parallelize, grid search scales poorly with the number of parameters d as the number of evaluations grows exponentially. It also uses computation on regions of the search space that have little influence on model performance, being very ineffective. Despite these limitations, grid search remains a valuable baseline. Comparison between grid search and random search can be seen in Figure 2.1.

2.2.3 Random Search

Random Search [Bergstra and Bengio 2012] was introduced as a more efficient alternative to grid search. Instead of testing every combination, it samples random configurations from distributions defined for each hyperparameter. For example, the learning rate might be drawn from a log-uniform distribution, while dropout probability might come from a uniform one. Formally, each trial draws $x_i \sim p(X)$ from the search distribution $p(X)$.

This approach focuses on covering the search space more broadly, in contrast to the regular sampling of Grid search. [Bergstra and Bengio 2012] showed that, for many models, only a few hyperparameters significantly affect performance — random search therefore reaches good results faster, with fewer evaluations. It is a simple approach which can be computed in parallel, and it is often used for modern ML applications. Comparison between random search and grid search can be seen in Figure 2.1.

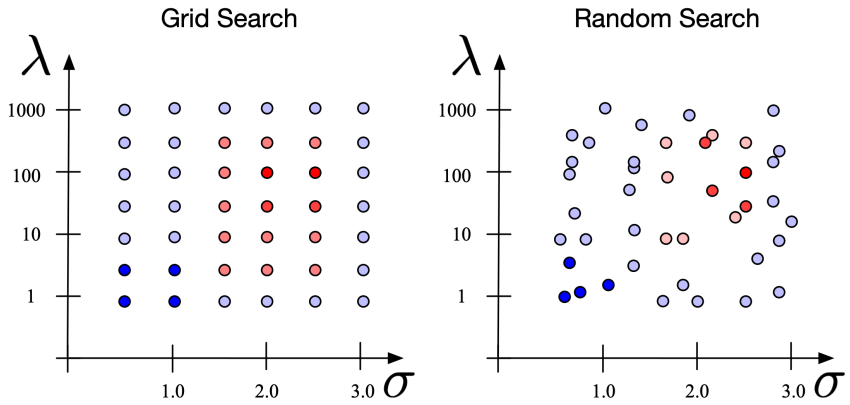


Figure 2.1: Comparison of Random Search and Grid Search strategies for hyperparameter optimization.[7]

.

2.2.4 Gaussian Processes

All definitions, notation, and descriptions in this section are derived from the foundational articles [Rasmussen and Williams 2006; Snoek, Larochelle, and Adams 2012; Bergstra, Bardenet, et al. 2011], unless stated otherwise.

A Gaussian Process (GP) is a non-parametric Bayesian model that defines a probability distribution over functions. GP is defined as a collection of random variables any finite number of which have a joint Gaussian distribution. We can write GP as:

$$f(x) \sim \mathcal{GP}(m(x), k(x, x')),$$

where $m(x) = \mathbb{E}[f(x)]$ is the mean function, and $k(x, x') = \mathrm{Cov}(f(x), f(x'))$ is the covariance, or *kernel*, function.

The mean function and the kernel function encode prior assumptions about the smoothness and structure of the unknown objective function. Given observed data, the GP provides a posterior distribution over functions that can be used to make predictions at unseen points along with uncertainty estimates. A visual representation of this process can be seen in Figure 2.2.

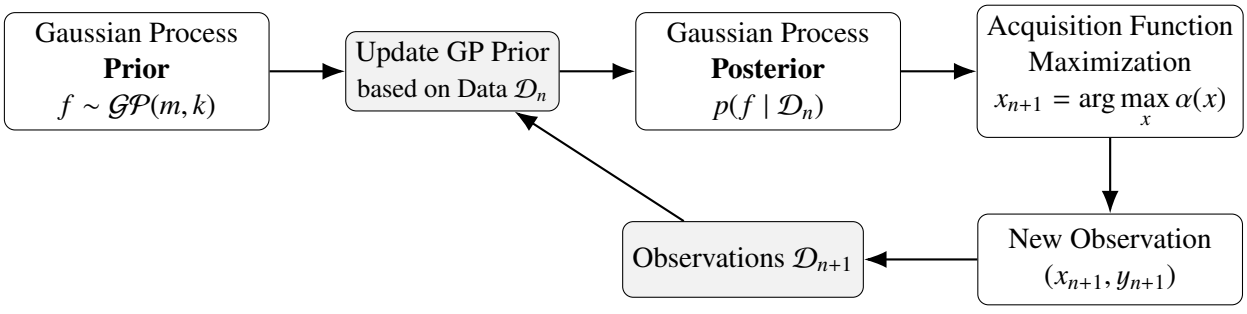


Figure 2.2: Bayesian optimization loop with Gaussian Process surrogate.

The Gaussian Process prior is fully specified by a mean function and a covariance (kernel) function.

[7] Image taken from `https://www.cs.cornell.edu/courses/cs4780/2023sp/lectures/lecturenote11`

The mean function encodes prior assumptions about the expected value of the function, and is often set to zero, as the kernel typically captures the dominant structural properties of the function. In practice, kernel choice has a far greater influence on model behavior than the mean function, particularly in Bayesian optimization settings.

Common choices for the covariance function include the squared exponential (RBF) kernel and the Matérn family of kernels. For two input points x and x', these kernels are defined as follows:

Squared Exponential Kernel (SE)

$$k_{\text{SE}}(x, x') = \exp\left(-\frac{\|x - x'\|^2}{2\ell^2}\right).$$

Matérn Kernel

$$k_{\text{Matérn-}\nu}(x, x') = \sigma^2 \frac{2^{1-\nu}}{\Gamma(\nu)} \left(\frac{\sqrt{2\nu}\,\|x - x'\|}{\ell}\right)^{\nu} K_\nu\left(\frac{\sqrt{2\nu}\,\|x - x'\|}{\ell}\right).$$

The parameter ℓ appearing in both kernels is known as the length-scale. It controls how rapidly the function values are allowed to vary with changes in the input: small values of ℓ permit rapid variation, while larger values enforce smoother, more slowly varying functions.

The squared exponential kernel is infinitely differentiable, which means that sample functions drawn from a GP prior using this kernel are extremely smooth. While this property can be advantageous in some settings, it may impose unrealistically strong smoothness assumptions in practical applications.

The Matérn kernel generalizes the squared exponential by introducing an additional smoothness parameter ν. This allows the Matérn class to represent functions with varying degrees of roughness: for example, $\nu = \frac{1}{2}$ yields a very rough exponential kernel, $\nu = \frac{3}{2}$ produces once-differentiable functions, and $\nu = \frac{5}{2}$ results in twice-differentiable functions.

As $\nu \to \infty$, the Matérn kernel converges to the squared exponential kernel, making the latter an infinitely smooth special case of the Matérn family. This flexibility has led the Matérn class to be widely recommended for modeling real-world processes, where assuming infinite smoothness may be inappropriate, while the squared exponential remains a popular default thanks to its simplicity and strong regularization properties.

2.2.4.1 Gaussian Process as a Surrogate Model

Bayesian optimization builds a **surrogate model** — an inexpensive statistical model that approximates the objective function $f(x)$ based on the results of previous trials. It predicts how good a given set of hyperparameters might be and also estimates the uncertainty of that prediction. At each step, the surrogate is updated using new observations, which allows the optimization algorithm to focus on the most promising regions of the search space. An effective choice for a surrogate model is Gaussian Process.

Computing predictive mean and variance After evaluating the objective at a set of points

$$\mathcal{D}_t = \{(x_i, y_i)\}_{i=1}^t, \quad \text{where } y_i = f(x_i) + \epsilon_i,$$

the GP updates its belief about $f(x)$ through conditioning on this data. $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$ denotes independent additive observation noise.

For a new point x, which has not been evaluated before the GP gives a predictive **mean** and **variance**:

$$\mu_* = \mathbf{k}_*^\top (K + \sigma_n^2 I)^{-1} \mathbf{y},$$
$$\sigma_*^2 = k(x_*, x_*) - \mathbf{k}_*^\top (K + \sigma_n^2 I)^{-1} \mathbf{k}_*,$$

where:

- K is the $n \times n$ covariance matrix with entries $K_{ij} = k(x_i, x_j)$,
- $\mathbf{k}_*$ is the covariance vector between the new point and training inputs, defined as $\mathbf{k}_* = [k(x_*, x_1), \ldots, k(x_*, x_n)]^\top$,
- σ_n^2 is the variance of the observation noise,
- $k(\cdot, \cdot)$ is the covariance function (kernel).

The mean $\mu_*(x)$ represents the GP's estimate of $f(x)$, while the variance $\sigma_*^2(x)$ measures how uncertain that prediction is. Together, these two quantities define a full probability distribution over possible outcomes, which the acquisition function uses to decide where to sample next.

Note for the notation: μ_* is sometimes written as $\mu_t(x)$. The notation $\mu_t(x)$ often appears in Bayesian optimization papers (such as [Snoek, Larochelle, and Adams 2012]) where the subscript t emphasizes the iteration number or the conditioning on data observed until iteration t, while μ_* notation is used by [Rasmussen and Williams 2006].

2.2.4.2 Acquisition Functions

The GP alone only predicts how good a configuration might be. To actually choose the next point to test, Bayesian Optimization uses an **acquisition function** $a(x \mid \mathcal{D}_t)$. The acquisition function scores each candidate point and tries to balance two goals:

- **Exploitation**: selecting hyperparameter settings that are expected to perform well, i.e., where the predicted mean $\mu_*(x)$ is low,
- **Exploration**: selecting regions of the search space where the model is uncertain, i.e., where the predicted variance $\sigma_*(x)$ is high.

The next point to evaluate is chosen as:

$$x_{t+1} = \arg\max_x a(x \mid \mathcal{D}_t).$$

The most common acquisition functions used in Bayesian Optimization are:

Probability of Improvement (PI): PI chooses points that are likely to outperform the best result so far. If $f(x_{best})$ is the best value found so far, then PI is:

$$a_{\text{PI}}(x) = P(f(x) \leq f(x_{best}) - \xi) = \Phi\left(\frac{f(x_{best}) - \mu_*(x) - \xi}{\sigma_*(x)}\right),$$

where $\Phi(\cdot)$ is the standard normal Cumulative Distribution Function (CDF) and $\xi \geq 0$ is a parameter controlling exploration. When $\xi = 0$, the method mostly performs trials in known good areas; larger ξ leads to more exploration.

Expected Improvement (EI): EI considers how big the improvement might be. It can be written as

$$a_{\mathrm{EI}}(x) = (f(x_{best}) - \mu_*(x) - \xi)\Phi(Z) + \sigma_*(x)\phi(Z),$$

where

$$Z = \frac{f(x_{best}) - \mu_*(x) - \xi}{\sigma_*(x)},$$

and $\phi(\cdot)$ is the standard normal Probability Distribution Function (PDF). This function gives high values to points that are either expected to perform well or are uncertain enough to be worth testing. EI is the most widely used acquisition function.

Upper Confidence Bound (UCB): UCB method combines the mean and uncertainty into a single formula:

$$a_{\mathrm{UCB}}(x) = \mu_*(x) - \kappa\,\sigma_*(x),$$

where $\kappa > 0$ is a parameter controlling the balance. A small κ means focusing on exploitation, while a larger value of κ leads to more exploration. UCB is simple and has strong theoretical properties in some settings.

2.2.4.3 Optimization Loop

As seen in Figure 2.2, Bayesian optimization works in an iterative loop:

1. Fit the GP surrogate to the current data.
2. Use the acquisition function to select the next candidate point.
3. Train and evaluate the model at that point to get a new observation.
4. Update the GP with this new data.

This loop continues until a stopping condition, such as a maximum number of trials or time limit, is reached. Example of GP can be seen in Figure 2.3.

2.2.4.4 Advantages

- **Strong modeling of parameter interactions**: Gaussian Processes define a joint probabilistic model over the objective function, allowing them to capture complex correlations between hyperparameters.
- **Sample efficiency**: by modeling predictive uncertainty, GP-based methods often achieve good performance with relatively few evaluations.

2.2.4.5 Limitations

- **Poor scalability**: Gaussian Process inference typically requires $O(n^3)$ time and $O(n^2)$ memory due to kernel matrix operations.
- **Limited support for categorical and conditional parameters**: standard GP models assume continuous inputs and struggle with discrete, categorical, or tree-structured search spaces without specialized kernels or encodings.
- **Reduced effectiveness in high-dimensional spaces**: as dimensionality increases, kernel-based similarity measures become less informative, often leading to slower convergence compared to methods designed for high-dimensional settings.

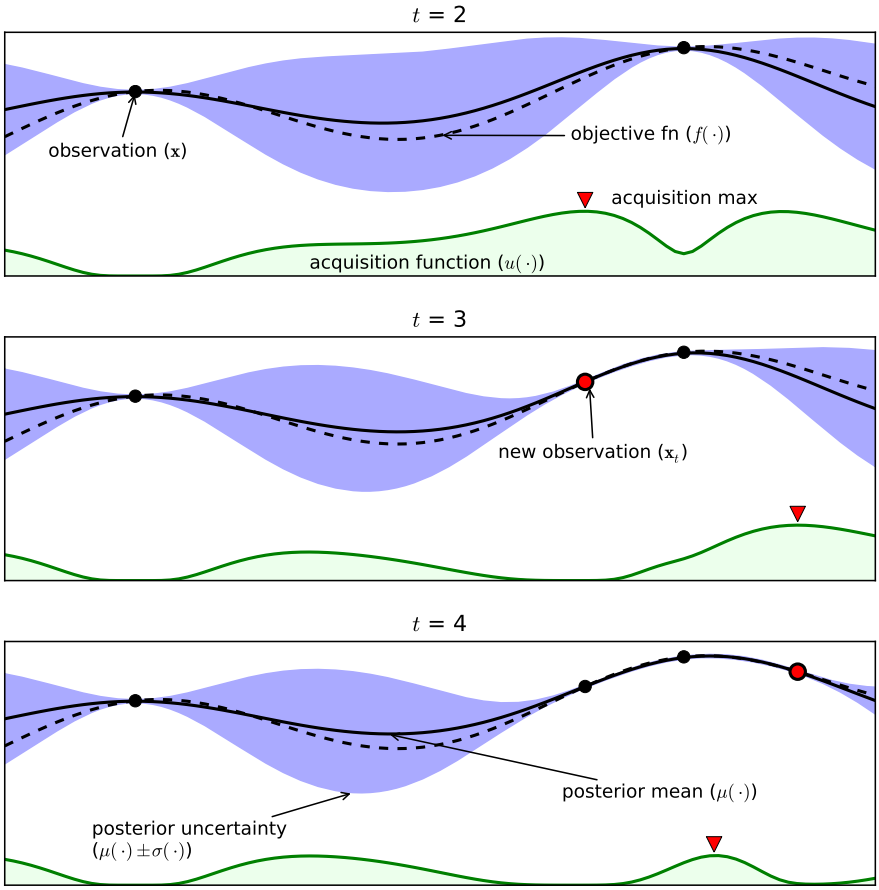


Figure 2.3: Illustration of Gaussian processes (image taken from [Brochu, Cora, and Freitas 2010]). For each iteration a surrogate model is fitted to the observed data. The new posterior mean and covariance are computed, and based on them the acquisition function selects a new point on which to evaluate the objective function.

2.2.5 Tree-structured Parzen Estimator

All definitions, notation, and descriptions in this section are derived from the foundational articles by Bergstra et al. [Bergstra, Bardenet, et al. 2011; Bergstra, Yamins, and Cox 2013], unless stated otherwise.

The Tree-structured Parzen Estimator (TPE) is an alternative to Gaussian Process-based Bayesian optimization. It was designed to handle complex hyperparameter spaces that include discrete, categorical, conditional, or hierarchical parameters — cases where Gaussian Processes are less effective. Instead of modeling the objective function $p(y \mid x)$ directly, TPE models the conditional probability density of hyperparameters given observed performance values.

Core Idea TPE first draws a given number of random configurations (e.g. 10), evaluates them and then computes the l and g densities based on the following division:

- **good** configurations that achieved results better than a chosen threshold y^*
- **bad** configurations that performed worse

It then estimates two probability distributions over the hyperparameter space:

$$l(x) = p(x \mid y < y^*), \qquad g(x) = p(x \mid y \geq y^*),$$

where:

- $l(x)$ captures the density of hyperparameters associated with good results, and
- $g(x)$ captures the density of hyperparameters associated with poor results.

The threshold y^* is typically chosen as a quantile of the observed performance values — for example, the 10th percentile of all previous losses. This means roughly 10% of the best-performing trials are used to estimate $l(x)$, while the remaining 90% form $g(x)$. This separation allows TPE to "learn what good looks like" and then actively search for new hyperparameters similar to those that already work well.

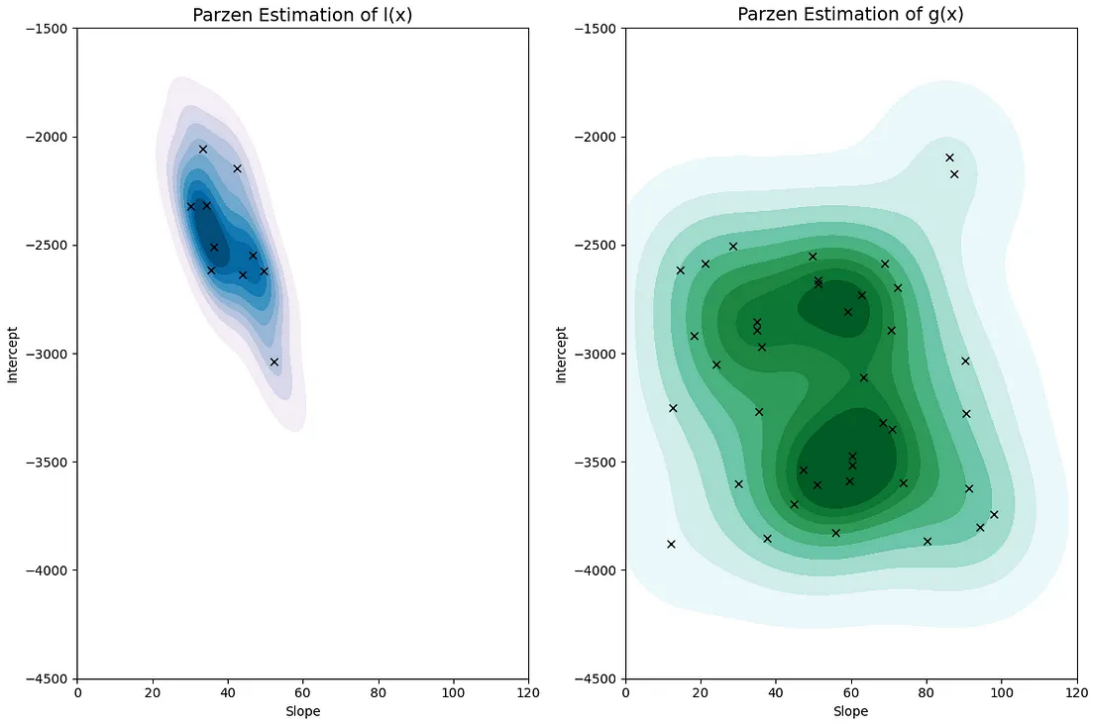


Figure 2.4: Illustration of the TPE density estimators $l(x)$ and $g(x)$.[8]

2.2.5.1 Density Modeling

Both $l(x)$ and $g(x)$ are modeled using **Parzen estimators**, a type of non-parametric kernel density estimator (KDE). The algorithm defines $l(x) = p(x \mid y < y^*)$ and $g(x) = p(x \mid y \geq y^*)$, where the threshold y^* is chosen as a quantile of observed losses.

Once $l(x)$ and $g(x)$ are estimated, the next candidate configuration x' is chosen by maximizing the ratio:

$$x' = \arg\max_x \frac{l(x)}{g(x)}.$$

[8]Image taken from https://medium.com/data-science/building-a-tree-structured-parzen-estimator-from-scratch-kind-of-20ed31770478

This means that TPE prefers hyperparameter settings that are highly probable under $l(x)$ (similar to successful past runs) but unlikely under $g(x)$ (dissimilar to unsuccessful runs).

To make this approach computationally efficient, each hyperparameter x_i is modeled individually rather than modeling all hyperparameters together. In probabilistic terms, the joint density $p(x \mid y)$ is factorized as the product of marginal densities:

$$p(x \mid y) = \prod_i p(x_i \mid y).$$

This means that the algorithm estimates the probability of each hyperparameter independently and then combines them, avoiding the complexity of modeling interactions between all hyperparameters at once.

This factorized approach also works with tree-structured search spaces, where some hyperparameters are only relevant when others take certain values. The algorithm handles these dependencies by treating the configuration space as a tree, so it can correctly model which hyperparameters are active in which cases.

For continuous hyperparameters, the marginal $p(x_i \mid y)$ is estimated using Gaussian kernels centered at observed values. For discrete or categorical hyperparameters, a simple histogram or categorical distribution is used.

The density estimation and sampling process can be summarized as:

1. Fit the two density models $l(x)$ and $g(x)$ from all collected trials.
2. Draw a large number of candidate samples $x_1, x_2, \ldots, x_N$ from $l(x)$.
3. Evaluate the ratio $l(x)/g(x)$ for each candidate.
4. Select the candidate with the highest ratio as the next configuration to evaluate.
5. Repeat.

2.2.5.2 Advantages

- **Handles both categorical and discrete parameters**
- **Conditional parameters**: supports tree-structured search spaces (e.g., only tune `beta1` and `beta2` if optimizer = Adam)
- **Scalability**: avoids the cubic complexity of Gaussian Processes and adapts well to many parallel trials.

2.2.5.3 Limitations

- **Weaker modeling of parameter interactions**: since TPE models each hyperparameter independently using one-dimensional density estimators, it may capture complex correlations between parameters less effectively than joint models such as Gaussian Processes.
- **Less sample-efficient in smooth, low-dimensional spaces**: in problems where the objective function is smooth and low-dimensional, GP-based Bayesian optimization may converge in fewer evaluations.

2.2.5.4 Implementation in Practice

TPE is widely adopted as the default sampler in modern hyperparameter optimization frameworks such as Optuna [Akiba et al. 2019]. In practice, Optuna maintains a history of past trials, selects a quantile for y^* (commonly 10–20%), updates the $l(x)$ and $g(x)$ density estimators after each evaluation, and draws candidate points from $l(x)$ to compute the likelihood ratios $l(x)/g(x)$. The configuration with the highest ratio is chosen for the next trial [Akiba et al. 2019].

The popularity of TPE stems from its flexibility in handling complex, hierarchical, and conditional search spaces, as well as its scalability to large optimization tasks. TPE's behavior is determined by several internal parameters — such as the quantile threshold for distinguishing good and bad observations, the bandwidth of the kernel density estimators, and the prior distribution assumptions for each hyperparameter. These tuning parameters can significantly affect both optimization stability and convergence speed [Watanabe 2023].

Chapter 3

Proposed Method

This work follows up on my Bachelor's thesis [Morovicsová 2024]. Similarly to the previous work, I focus on the problem of automated translation of hieroglyphs. Specifically, since the dataset I have available contains hieroglyphs and their transliterations, I focus on the transliteration part of the translation (see Section 2.1.3.1).

In [Morovicsová 2024], I compared several types of neural networks to determine which method was most suitable for the task of automated translation of hieroglyphs. The results showed that the transformer architecture performed best (compared with simple RNN and LSTM architectures). Therefore, I continue with using the transformer architecture.

Dataset - in each of the experiments, the input data for the neural network are sentences encoded in Gardiner codes with the words already separated, and the output is their transliteration. For information about the dataset used, see Section 2.1.4.

This section further outlines the model architecture used and the choice of metrics. Then it moves on to the experiments - **Comparison of optimizers**: using 4 different optimizers for transformer hyperparameter tuning and comparing their performance and **Dataset size comparison**: estimating how much data is needed for achieving meaningful results. Finally, this chapter contains a brief section outlining the implementation.

3.1 Model Architecture

The core of the system is a standard Transformer encoder–decoder architecture introduced in [Vaswani et al. 2017]. It relies on multi-head self-attention, position-wise feed-forward layers, and positional encodings to model long-range dependencies without recurrence. For this work, I use the same implementation used in my Bachelor's thesis [Morovicsová 2024].

The model is compiled using the Adam optimizer, with categorical cross-entropy as the loss function. Accuracy is tracked as the primary evaluation metric during training.

3.2 Evaluation: Choice of metrics

To evaluate the performance, several standard metrics that capture different aspects of accuracy were used.

The two most common metrics used in training and evaluating machine learning translation models are **Accuracy** and **Loss**. Accuracy measures the proportion of correctly predicted tokens compared to the total number of predictions, while loss quantifies the difference between the model's predicted

probabilities and the true target distribution. They are connected in such a way that minimizing the loss function during training typically leads to higher accuracy.

In addition to overall accuracy, I use **word-level** and **character-level** accuracy to capture more fine-grained information about translation quality. Word accuracy counts the percentage of correctly translated words (with no errors), whereas character accuracy counts the percentage of correctly translated individual characters.

The **Levenshtein distance** (also known as edit distance) measures the minimal number of single-character edits—insertions, deletions, or substitutions—required to transform one string into another. It provides a quantitative measure of similarity between the predicted and ground truth translation.

For this work, the specific implementation of the Levenshtein distance done by Rosmorduc was used [Rosmorduc 2020]. It includes specific adaptations for the Ramses dataset to handle the characteristics of the corpus.

Furthermore, the **Gold standard** is used to represent the number of sentences translated with no errors at all - sentences that are identical to the reference translation.

The **ROUGE-L** metric measures similarity between the predicted and reference translation using the Longest Common Subsequence - the longest sequence of tokens that appears in both texts in the same order (not necessarily next to each other). Compared to exact-match accuracy, ROUGE-L gives credit when the prediction keeps the same word order even if a few words are missing or extra [C.-Y. Lin 2004].

The **sacreBLEU** score is a standard way to compute BLEU so results are easier to compare, because it uses fixed tokenization and reporting. BLEU measures translation quality using modified n-gram precision (often up to 4-grams) and a brevity penalty to avoid very short outputs. Using sacreBLEU helps avoid different BLEU settings giving different scores [Papineni et al. 2002; Post 2018].

3.3 Comparative Study Of Hyperparameter Optimizers

This section describes the experiment used to evaluate different hyperparameter optimization strategies for the hieroglyphic transliteration task. I compare four optimizers: Grid Search, Random Search, TPE and Gaussian Processes. The underlying theoretical concepts of these optimizers are introduced in Chapter 2. The aim of this study is to compare these four optimization methods and determine which is most effective for training a Transformer-based model for Ancient Egyptian transliteration.

Hypothesis: Bayesian optimization methods (TPE and Gaussian Processes) will reach high-quality hyperparameters more efficiently (i.e., with fewer trials) than non-Bayesian approaches such as Random and Grid Search.

An additional goal is to compare the systematically chosen hyperparameters with those found manually in my previous work [Morovicsová 2024] and examine which hyperparameters for the transformer model yield the best results.

Furthermore, a study comparing the stochastic optimizers was conducted to evaluate the stability of TPE and GP by comparing their performance across different random initializations. Additionally, since prior work [Morovicsová 2024] indicated that excessively large models relative to the dataset size may struggle to learn effectively, I conducted a focused comparison of model size versus performance using the models trained in previous experiments.

The detailed results of the experiments are presented in Section 4.2.

3.3.1 Experiment Setup

The experiment has the following parts:

1. **Small-scale experiment**: running all optimizers on a tiny subset of the Ramses corpus (first 5 sentences) to validate the correctness of the implementation and the optimizer behavior.
2. **Main Hyperparameter Search**: running 50 trials for each optimizer on 30% of the dataset, using the fixed search space in Table 3.1.
3. **Full model training**: retraining the best hyperparameter configuration found in Main Hyperparameter Search on the entire dataset, and then comparing it with the manually found model in previous work [Morovicsová 2024].
4. **Model Size vs Model Performance Comparison**: Short comparison of model size vs model performance.
5. **Comparing Stochastic Optimizers**: A robustness study evaluating stability of TPE and GP over 100 runs with different random seeds.

All experiments use the same baseline Transformer model as described in Section 3.1. The dataset used is described in Section 2.1.4.

For the main experiment, 30% of the original dataset was used. This subset size represents a trade-off between computational cost and representativeness of the data. Using a sufficiently large portion of the dataset allows the model to capture the main patterns of the task while still keeping training time manageable during hyperparameter optimization. Similar approaches, where a subset of the available data is used for efficient model selection before full training, are commonly suggested in the literature (e.g., [Bergstra and Bengio 2012]).

All optimization algorithms are executed through the Optuna framework with a fixed search space (since Grid search requires fixed search space). The selected hyperparameter values to explore are given in Table 3.1. These values were selected for simplicity, to limit the size of the search space ensuring fair comparison between optimizers. Using the same search space ensures a fair comparison between optimizers, as differences in performance are due to the optimization strategy rather than to differences in the available configurations.

Hyperparameter	**Values**
h (number of heads)	2, 4, 8
d_k (key dimension)	32, 64
d_{ff} (feedforward layer size)	512, 1024, 2048
d_{model}	256, 512
n (number of layers)	2, 4, 6

Table 3.1: Hyperparameter search space used in the experiments

For each suggested hyperparameter configuration, the model is either retrieved from a cache (if an identical configuration has already been evaluated) or newly trained for 20 epochs. The caching mechanism avoids redundant training and significantly reduces computational cost when different optimizers suggest the same configuration.

Note on the choice of the number of epochs Each trial serves as a preliminary evaluation of the proposed hyperparameter configuration rather than as full model training. The choice of 20 epochs is motivated by efficiency: in previous work, models were trained for a larger number of epochs, but for hyperparameter selection it is sufficient to observe early performance trends. Only the best-performing configurations are then trained further on the full dataset for a larger number of epochs.

Note on the caching mechanism The cross-study caching mechanism works as follows: for each suggested hyperparameter configuration, the optimizer first checks whether this configuration has already been evaluated in a previous study. If so, the cached result is reused. This approach leads to a substantial reduction in computational cost.

For the caching mechanism to be fully reliable, the training process must be completely reproducible. This requires the use of a fixed random seed. Before running the main experiments, all relevant random seeds were explicitly set, and reproducibility was verified by running the same experiment multiple times and obtaining identical results. This confirmed that the fixed seed was correctly applied and that no remaining sources of randomness affected the training process.

To ensure full reproducibility, the fixed seed was set in the following places:

```
os.environ["PYTHONHASHSEED"] = str(seed)
np.random.seed(seed)
random.seed(seed)
tf.random.set_seed(seed)
tf.config.experimental.enable_op_determinism()
```

Evaluation: The experiment uses two levels of evaluation:

- **Trial metric**: Validation accuracy used during hyperparameter optimization.
- **Final evaluation**: Word accuracy, character accuracy, Levenshtein distance, ROUGE-L, SacreBLEU calculated on the test dataset.

3.4 Dataset Size Dependency Study

Understanding how dataset size impacts model performance is critical, especially for tasks such as Ancient Egyptian transliteration, where data is scarce and costly to obtain. Unlike many high-resource language applications, the training data here has to be manually translated by highly specialised experts, making large-scale data collection difficult. This experiment investigates how increasing amounts of data influence model outcomes and helps identify the minimum dataset size required to achieve effective results. Such insights are important for determining whether neural approaches are suitable or if alternative methods might be more practical when data are limited.

To investigate the influence of dataset size on model performance, I conducted an experiment using the Ramses corpus (see Section 2.1.4), which contains approximately 67,000 sentences. The corpus was split into 67 progressively larger subsets: the first containing the initial 1,000 sentences, the second 2,000, and so forth, up to the full dataset.

Hypothesis : **The plateau hypothesis** - a concept often described in academic literature as diminishing returns or data saturation - suggests that as the amount of training data increases, model performance will improve up to a point but eventually level off, indicating diminishing returns. Identifying this plateau is important because it reveals the minimum dataset size needed to achieve good results. This helps guide resource allocation, especially in low-resource settings where data collection is costly or limited.

For results of this experiment, refer to Section 4.3.

3.4.1 Experimental Setup

The experiment has the following parts:

1. **Subset construction**: splitting the full Ramses corpus into 67 progressively larger training subsets, starting at 1,000 sentences and increasing up to the full dataset size. Each subset contains all sentences from the previous subset plus additional samples, so the training size grows in a controlled way.

2. **Initial small-model sweep**: training a deliberately small Transformer model on every subset (67 dataset subsets) for up to 20 epochs. For each run, validation accuracy and validation loss are recorded to estimate where performance begins to plateau and to confirm that the training pipeline works correctly.

3. **Main comparison on selected sizes**: training larger models only on selected subset sizes (in thousands of sentences: $1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 19, 25, 31, 37, 43, 49, 55, 61, 67$). Not training on every subset reduces training cost while still showing how dataset size affects performance across different models.

4. **Dependency on Initial Conditions**: Evaluating the model stability based on different random initializations.

All experiments use the same baseline Transformer architecture described in Section 3.1. To keep results comparable across all runs, every model is evaluated on the same fixed test dataset.

Evaluation: For the initial sweep, validation accuracy and validation loss are tracked for each subset. In the main comparison, I additionally report the average Levenshtein distance per character.

3.5 Implementation

The codebase builds on work originally developed during my bachelor's thesis [Morovicsová 2024]. It uses Optuna [Akiba et al. 2019] as the hyperparameter optimization framework and relies on Optuna's optimizer implementations with their default settings. The transformer model itself was implemented using Keras 3.

Optuna provides standardized implementations of all optimizers used in this work, exposing a small set of user-configurable parameters while handling the remaining logic internally. Since the conceptual mechanisms of these methods have already been discussed in Chapter 2, the following description focuses only on their practical configuration within Optuna.

For the **Tree-Parzen Estimator (TPE)**, Optuna uses separate density estimators for good and bad trials. Its main tunable parameters are:

```
n_startup_trials   # default: 10     (number of initial random trials before TPE starts)
gamma              # default: 0.25   (fraction of best trials for "good" model)
n_ei_candidates    # default: 24     (number of candidates sampled for EI)
```

For the **Gaussian Process (GP)** sampler, Optuna uses Matérn kernel with the acquisition function being log Expected Improvement. Its main tunable parameters are:

```
n_startup_trials   # default: 10     (number of initial random trials before fitting GP)
gpr_alpha          # default: 1e-10  (added to kernel diagonal for stability)
```

The **Random Search** sampler and **Grid Search** expose only minimal configurations, such as seed and search space.

In this work, all optimizers were used with these default settings.

Chapter 4

Experiment results

This chapter presents the experimental evaluation of the proposed transformer-based method for hieroglyphic transliteration. The goal of these experiments is to evaluate the efficiency and robustness of hyperparameter optimization methods and to analyze the relationship between dataset size and transliteration performance.

Comparison of hyperparameter optimization methods: Four different optimization methods: Grid Search, Random Search, Tree-Parzen Estimator (TPE), and Gaussian Process (GP) are compared to evaluate their effectiveness when tuning a Transformer model. The aim is to determine which optimizer performs best for this task (transliteration of hieroglyphs) and to compare the best set of hyperparameters found automatically with those found in previous work.

Dataset size dependency: This experiment investigates how the amount of training data affects model performance and estimates how much data is needed to achieve stable and meaningful results.

All experiments use the same Transformer architecture and a fixed hyperparameter search space to ensure fair comparison. The evaluation metrics used are described in Section 3.2. The experimental setup and model design are described in detail in Chapter 3. All experiments are carried out on the Ramses dataset (see Section 2.1.4).

4.1 Hardware Used

The majority of the experiments were conducted on the Helios server (`http://helios.fjfi.cvut.cz/`), using GPU-enabled compute nodes. When the Helios job queue was saturated or for smaller-scale experiments, training was performed locally on my personal computer, which has an NVIDIA GPU and 32 GB of RAM.

The maximum RAM required for the large-scale training experiments was 64 GB, while most experiments ran sufficiently with 16 GB or 32 GB.

4.2 Results: Comparative Study Of Hyperparameter Optimizers

As described in Section 3.3, this experiment evaluates the performance of different hyperparameter optimization methods for the task of Ancient Egyptian transliteration. The main objective is to determine which optimization method is able to find the best-performing Transformer model within the smallest number of trials, and therefore with the lowest computational cost.

In addition, the results are compared with the model obtained in my previous work, where hyperparameters were selected manually using a trial-and-error approach. This comparison aims to determine

whether the systematic optimization methods can reproduce the previously selected model or identify a model with better performance.

4.2.1 Implementation details

All optimization experiments are conducted using the Optuna framework, which provides unified implementations of all compared optimization methods. To ensure reproducibility, fixed random seeds are used consistently for model initialization and for the optimizers, allowing the entire experiment to be repeated under identical conditions.

The experiments are based on the code developed during my Bachelor's thesis [Morovicsová 2024]. The original implementation was initially written using `Keras 2` and later partially migrated to `Keras 3`, without fully utilizing several features provided by newer versions of Keras.

For the purposes of this work, the codebase was extended and improved in several ways. The following enhancements were made at the Keras training level:

- **Added early stopping** to reduce overfitting.
- **Option to restore the best model** observed during training.
- **CSV logging** to record training and validation metrics (previously done using a custom dictionary structure).
- **Model checkpointing** to save model weights during training without breaking the training loop.

In addition to these framework-specific changes, several general improvements were implemented to support reliable experimentation:

- **Improved random seed control** to ensure consistent and reproducible results across runs.
- **Integration with Optuna** to enable automated hyperparameter optimization.
- **A cross-study caching mechanism** to speed up training and avoid repeated evaluation of identical model configurations.

The code is available on Github: `https://github.com/casualMathEnjoyer/Hier`.

4.2.2 Optimizers: Preliminary Experiment

To test the implementation for possible errors, I first ran the optimization on a small subsection of the dataset. From the entire Ramses dataset I selected the first 5 sentences and used them both as the training and validation dataset. The goal of this experiment was not only to ensure that the entire optimization pipeline works as intended, but to also look at the results and ensure that they make sense.

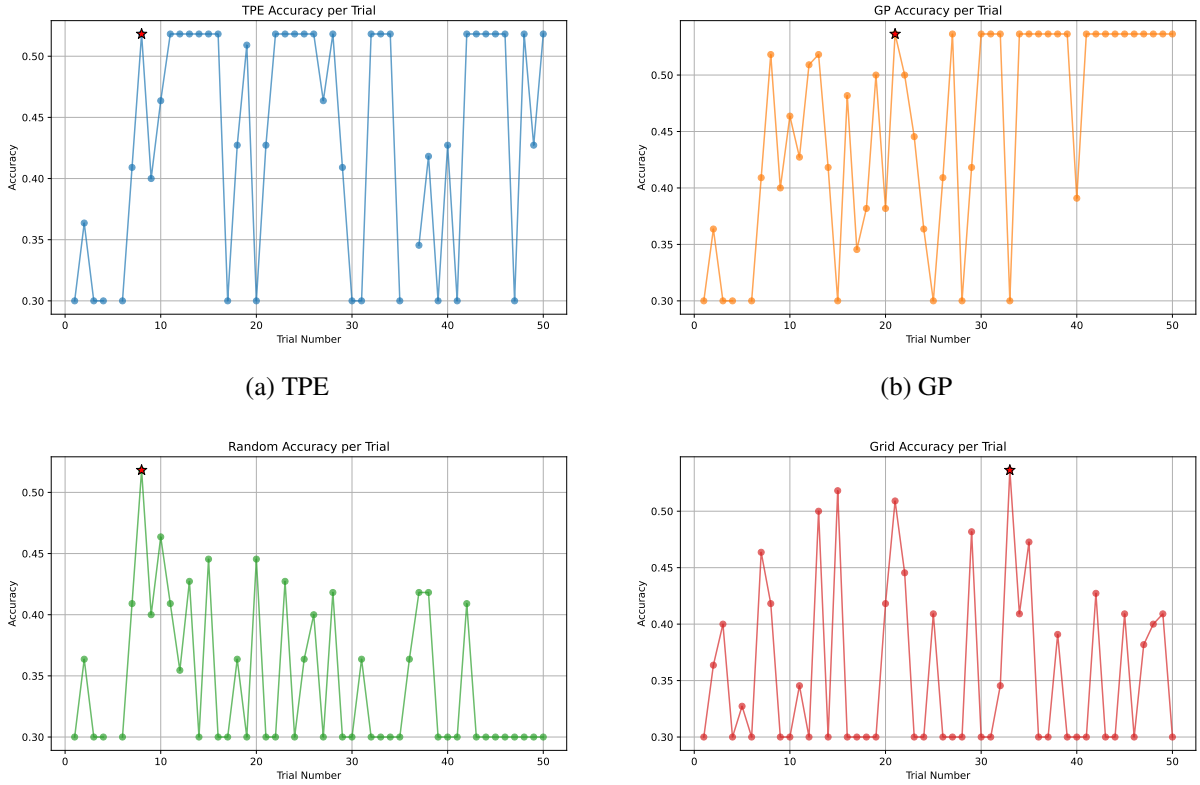
Each model was trained for 20 epochs, with the early stopping mechanism active. Each optimizer was run for a fixed number of 50 trials. All optimizers were initialized with the same fixed seed.

4.2.2.1 Performance of individual optimizers

The behavior of the different optimizers is shown in Figure 4.1. The graphs indicate that some optimizers repeatedly find models with the same accuracy values, either the same high accuracy or the same or very similar low accuracy. This issue is discussed in more detail in Section 4.2.3.2.

An interesting pattern can be observed in the results of Random Search. This is unexpected because Random Search does not use information from previous evaluations and selects hyperparameters randomly. This pattern can be explained by two main factors. First, Random Search may have found good configurations early by chance, which resulted in higher accuracy at the beginning. Second, as the search continued, Random Search increasingly selected configurations that were too complex for the available dataset (only 5 sentences). These overly complex models failed to train properly. Since the same random seed was used in all experiments, these failing models consistently reached the same low accuracy of 0.3000. This created the appearance of a systematic downward trend, even though Random Search does not learn from previous evaluations.

In contrast, the TPE and GP optimizers show a clear upward trend, which is expected because they adapt their search based on past results. Finally, as expected, Grid Search does not show any clear performance trend.



(a) TPE

(b) GP

(c) Random

(d) Grid

Figure 4.1: Validation accuracy per trial number for each optimizer (Preliminary Experiment).

4.2.2.2 Summary of results

Table 4.1 reports the best accuracy achieved by each optimizer, the trial number at which this accuracy was first reached, and the corresponding hyperparameters. Notably, both the GP and Grid optimizers achieved the highest accuracy of 0.5364, with GP reaching this performance earlier (at trial 21) compared to Grid (at trial 33). TPE and Random both achieved an accuracy of 0.5182 at trial number 8. In this experiment, the TPE optimizer didn't find the best model ($h = 2$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 512$).

Optimizer	Accuracy	Trial Number	Hyperparameters
TPE	0.5182	8	$h = 4, d_k = 32, d_{ff} = 512, d_{model} = 256, n = 2$
GP	0.5364	21	$h = 2, d_k = 32, d_{ff} = 512, d_{model} = 512, n = 2$
Random	0.5182	8	$h = 4, d_k = 32, d_{ff} = 512, d_{model} = 256, n = 2$
Grid	0.5364	33	$h = 2, d_k = 32, d_{ff} = 512, d_{model} = 512, n = 2$

Table 4.1: The best accuracy per optimizer, trial number it was first achieved, and corresponding hyperparameters

4.2.2.3 Summary of Best-Performing Models

For comparison, the model I found by handpicking the hyperparameters for my Bachelor's thesis [Morovicsová 2024] had the following hyperparameters: $h = 2, d_k = 64, d_{ff} = 512, d_{model} = 256, n = 2$. This model is different from any of the ones found by the hyperparameter search on this small dataset.

Table 4.2 presents the models that achieved an accuracy above 0.5. These models demonstrated promising performance and would normally be selected for full training on the complete training dataset, followed by evaluation on the test dataset. However, since this experiment was intended primarily as a small-scale validation of the code and workflow, I did not proceed with full training at this stage. The main objective here was to verify that all components of the pipeline function correctly. Having confirmed this, I can move on to the main experiment, which will involve higher computational requirements.

Accuracy	**Configuration**
0.5364	$h = 2, d_k = 32, d_{ff} = 512, d_{model} = 512, n = 2$
0.5182	$h = 2, d_k = 64, d_{ff} = 512, d_{model} = 256, n = 2$
0.5182	$h = 4, d_k = 32, d_{ff} = 512, d_{model} = 256, n = 2$
0.5091	$h = 4, d_k = 32, d_{ff} = 512, d_{model} = 512, n = 2$
0.5000	$h = 2, d_k = 32, d_{ff} = 512, d_{model} = 256, n = 2$
0.5000	$h = 8, d_k = 32, d_{ff} = 512, d_{model} = 256, n = 2$

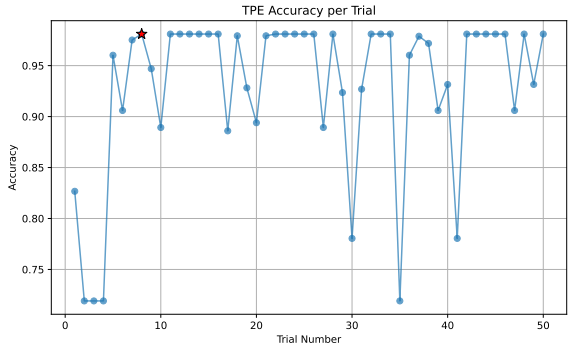
Table 4.2: Top trials sorted by validation accuracy, trained as part of the preliminary experiment, on dataset containing only 5 sentences. The best model found by manual tuning as part of my Bachelor's thesis is highlighted.

4.2.3 Optimizers: Main Experiment

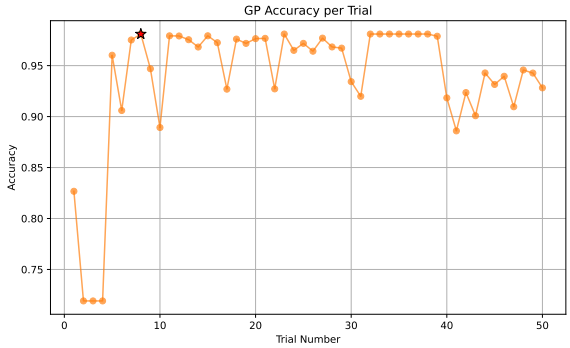
After verifying that the optimization process worked correctly on the smaller subset of data, I moved on to the main experiment using a larger portion of the dataset. When using hyperparameter search methods, the whole dataset is not usually used, as that would require a large amount of resources (especially in machine learning tasks). Sources such as [Bergstra and Bengio 2012] suggest using from 10% to 30% of the available data, depending on the size of the dataset. I decided to use 30% of the original training data. Each experiment was run similarly to the small experiment for 50 trials, with each optimizer exploring the hyperparameter space defined in Table 3.1. The same caching mechanism and early stopping were applied, ensuring that previously trained models were reused when possible, and training of individual models was stopped once no improvement was observed in validation accuracy. Each model was trained for 20 epochs and all optimizers were initialized with the same fixed seed.

4.2.3.1 Performance of individual optimizers

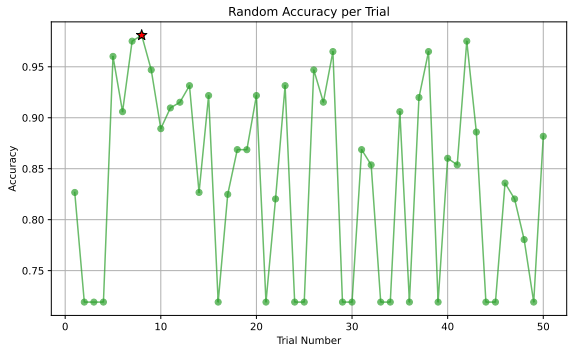
The results of each optimizer's performance during the main experiment are in Figure 4.2, showing the validation accuracy achieved per trial for each search strategy. Three of the optimizers, TPE, GP and Random search, all found the best model on trial number 8. This is due to the first 10 trials of these optimizers being chosen in the same way - both TPE and GP use the Random search for selecting the first trials and only after the first n random trials does the smart optimization start for TPE and GP. Thanks to using the same fixed seed, the results on the first 10 trials match exactly. We can observe that all four methods achieve good performance, with maximum validation accuracies exceeding 0.95.



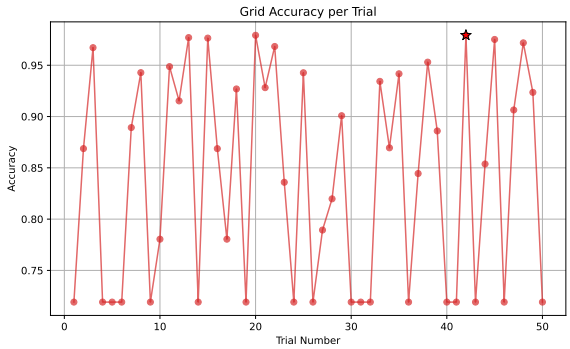
(a) TPE



(b) GP



(c) Random



(d) Grid

Figure 4.2: Validation accuracy per trial number for each optimizer (Main experiment).

The distribution of accuracy values in Figure 4.3 illustrates differences between methods: Bayesian methods show a tighter clustering of high-accuracy trials, while Random and Grid show broader distributions. This behaviour is expected because Bayesian optimizers use information from previous trials to focus on promising regions of the search space, leading to more consistent high performance, with occasional outliers corresponding to highly explorative trials. In contrast, Random and Grid search explore the space more uniformly.

4.2.3.2 Analysis of Identical Accuracy Values

During the hyperparameter optimization process, an interesting phenomenon was observed: multiple different model configurations achieved identical validation accuracy values. This occurred in two distinct scenarios with different underlying causes.

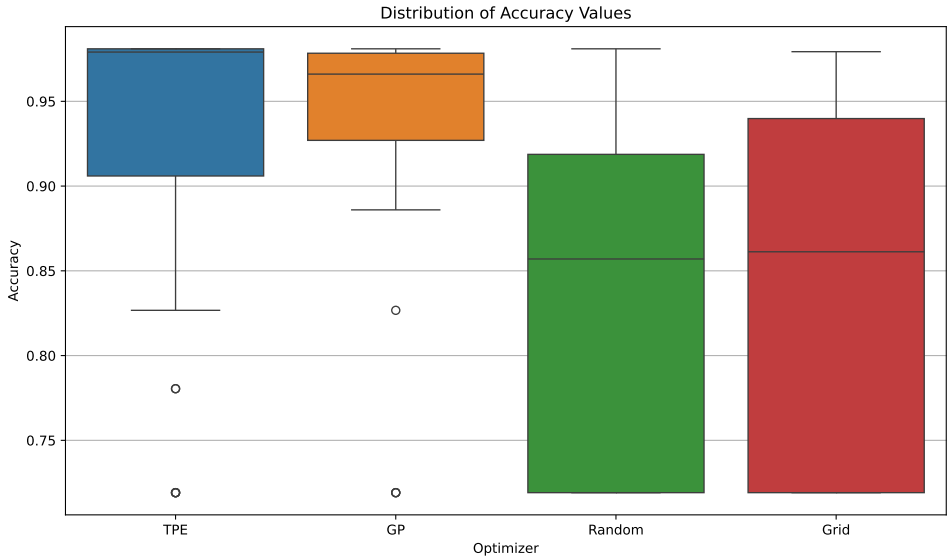


Figure 4.3: Distribution of achieved validation accuracies for each optimizer (main experiment).

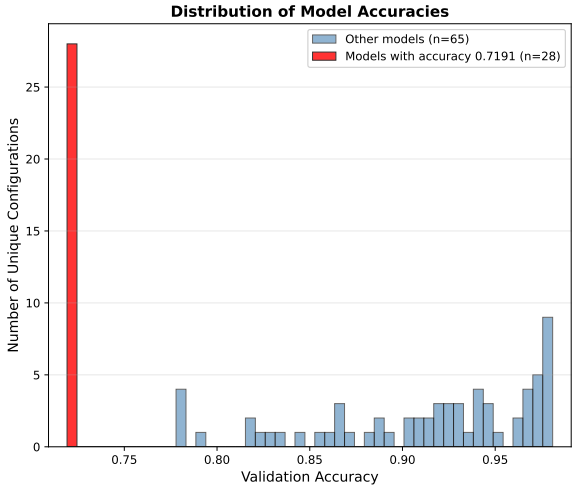
First, successful model configurations were frequently rediscovered by the same optimization algorithms. For instance, the best-performing configuration ($h = 4$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$) with validation accuracy of 0.9810 was found 33 times across all optimizers, with TPE discovering it 23 times and GP finding it 9 times. This behavior makes sense for Bayesian optimization methods like TPE and GP, which exploit promising regions of the hyperparameter space and tend to converge toward optimal configurations. Since this experiment used a fixed search space, the number of possible model configurations was limited. As a result, in some regions of the search space there were no alternative configurations close to the optimum, causing the optimizers to select the same model repeatedly.

Second, and more concerning, was the observation that 28 different model configurations all achieved the same low validation accuracy of 0.7191, as shown in Figure **??**. Analysis of these configurations revealed a clear pattern: 75% of these models had $n = 6$ layers (the maximum number in the search space), 64% had $d_{model} = 512$ (the largest model dimension), and 43% had $d_{ff} = 2048$ (the largest feedforward dimension). These models represent the most complex configurations in the search space, likely exceeding the capacity that can be effectively trained on the 30% dataset subset used for optimization.

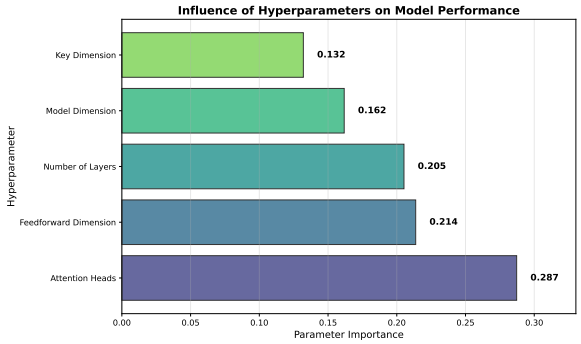
The identical accuracy values for these failing models can be attributed to two factors. First, all models were trained with the same fixed random seed (42), which means identical weight initialization and random operations during training. For models that fail to learn effectively, this deterministic setup causes them to converge to the same poor local minimum or baseline performance. Second, the early stopping mechanism, configured to monitor validation accuracy starting from epoch 3 with a patience of 5, quickly terminates training when no improvement is observed. Since these overly complex models fail to improve from the beginning, they all trigger early stopping at similar points, resulting in identical final accuracies.

To understand which hyperparameters have the strongest influence on model performance across all optimization trials, parameter importance analysis was performed using Optuna's fANOVA [Hutter, Hoos, and Leyton-Brown 2014]. This analysis combines all trials from TPE, GP, Random, and Grid optimizers into a single study and computes the importance of each hyperparameter by analyzing how much variation in validation accuracy can be attributed to changes in that parameter. The importance values are normalized such that they sum to 1, with higher values indicating that the hyperparameter has

a stronger influence on model performance. The results, shown in Figure **??**, reveal that attention heads (h) has the highest importance (0.29), followed by feedforward dimension (d_{ff}, 0.21) and number of layers (n, 0.21), while key dimension (d_k) shows the lowest influence (0.13).



(a)



(b)

Figure 4.4: (a) Distribution of unique model configurations by validation accuracy, showing a cluster of 28 configurations with the same low accuracy of 0.7191. (b) Parameter importance analysis computed across all optimization trials.

The parameter importance analysis shows that the number of attention heads (h) has the strongest effect on model performance, followed by the feedforward dimension (d_{ff}) and the number of layers (n). However, the consistent failure of models with very high complexity—specifically $n = 6$ layers, $d_{model} = 512$, and $d_{ff} = 2048$—shows that even though these parameters are not the most important overall, their extreme values often produce models that are too large to be trained successfully on the available dataset.

4.2.3.3 Summary of results

Table 4.3 summarizes the best validation accuracy, the trial number where it was first achieved, and the corresponding hyperparameter configuration for each optimizer.

Optimizer	Best Accuracy	Trial Number	Hyperparameters
TPE	0.9810	8	$h = 4, d_k = 32, d_{ff} = 512, d_{model} = 256, n = 2$
GP	0.9810	8	$h = 4, d_k = 32, d_{ff} = 512, d_{model} = 256, n = 2$
Random	0.9810	8	$h = 4, d_k = 32, d_{ff} = 512, d_{model} = 256, n = 2$
Grid	0.9793	42	$h = 4, d_k = 32, d_{ff} = 1024, d_{model} = 256, n = 2$

Table 4.3: Best validation accuracy per optimizer, trial number when it was first achieved, and corresponding hyperparameters of the model (main experiment).

Overall, the main experiment confirms that Bayesian optimization methods (TPE and GP) are the most effective for this task. They achieve high accuracy quickly and maintain stable performance across trials, whereas Random Search and Grid Search are less efficient and show greater variability. This is

expected because Bayesian optimizers use previous results to guide the search toward better regions of the hyperparameter space, while Random and Grid search do not.

4.2.3.4 Summary of Best-Performing Models

The best validation accuracy achieved in the main experiment was 0.9810, found by TPE, GP and Random optimizers with hyperparameters $h = 4$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$.

Other model configurations with accuracy ≥ 0.975 are mentioned in Table 4.4. The model previously found in my Bachelor's thesis ($h = 2$, $d_k = 64$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$) is also found, with validation accuracy 0.9765.

These models demonstrated promising performance and could be selected for full training on the complete training dataset. However, for the full model training, I decided to only train the best model for simplicity.

Accuracy	Configuration
0.9793	$h = 4$, $d_k = 32$, $d_{ff} = 1024$, $d_{model} = 256$, $n = 2$
0.9792	$h = 8$, $d_k = 32$, $d_{ff} = 1024$, $d_{model} = 256$, $n = 2$
0.9791	$h = 8$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$
0.9788	$h = 8$, $d_k = 64$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$
0.9770	$h = 2$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$
0.9768	$h = 2$, $d_k = 64$, $d_{ff} = 1024$, $d_{model} = 256$, $n = 2$
0.9765	$h = 2$, $d_k = 64$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$
0.9760	$h = 4$, $d_k = 64$, $d_{ff} = 1024$, $d_{model} = 256$, $n = 2$
0.9754	$h = 4$, $d_k = 64$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$
0.9752	$h = 8$, $d_k = 32$, $d_{ff} = 2048$, $d_{model} = 256$, $n = 2$

Table 4.4: Near-optimal trials sorted by validation accuracy (threshold ≥ 0.975). The previous best model found by manual tuning is highlighted.

4.2.3.5 Full model training

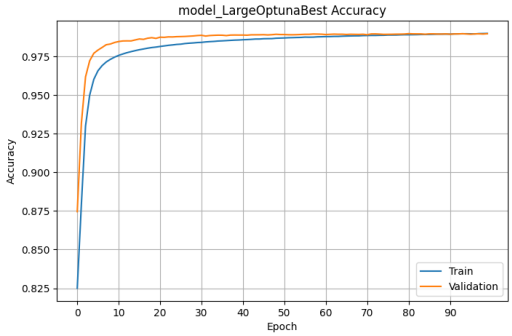
The best model identified through Optuna optimization on the limited subset of the data (30% of the full dataset) had the following hyperparameters: $h = 4$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, and $n = 2$. To complete the evaluation, I trained this model on the full dataset. Figure 4.5 presents the training and validation accuracy and loss across 100 epochs. Based on the convergence observed in these curves, no further training is necessary.

The model was trained on faculty server Helios, and the model training took approximately 55 hours.

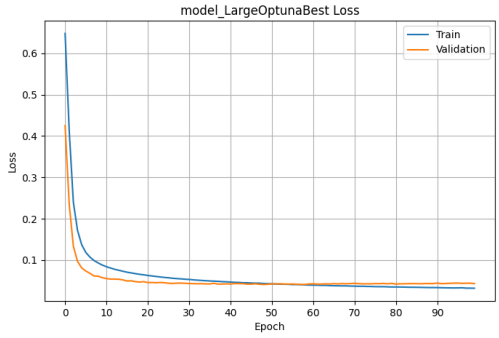
Results on the testing dataset can be seen in Table 4.5, where the model found by Optuna (OptunaBest) is compared with the best model architecture I found in my Bachelor's thesis (PreviousBest). I retrained the model from Bachelor's thesis from scratch as part of this experiment.

The resulting metric values differ slightly from those reported in the earlier work. This difference is due to a change in how the Levenshtein distance is computed. In the Bachelor's thesis, the metric was calculated on the full sentence, including the beginning-of-sentence and end-of-sentence tokens. In the current setup, these tokens are removed before evaluation, which generally leads to slightly higher (worse) Levenshtein distances.

The model found through automatic hyperparameter tuning achieves results that are comparable to, and in some metrics slightly better than, the model previously found through manual tuning. As shown



(a) Training and validation accuracy



(b) Training and validation loss

Figure 4.5: Full training results of the best model found by Optuna ($h = 4$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$), showing training and validation accuracy and loss over epochs.

in Table 4.5, the automatically selected model achieves a slightly lower average Levenshtein distance (0.0993 vs. 0.1010) and higher character-level accuracy (61.50% vs. 61.21%). It also reaches marginally higher ROUGE-L and SacreBLEU scores. The manually tuned model performs slightly better in word-level accuracy and produces a higher number of fully correct (gold) predictions.

Although the absolute metric values may seem low when compared with modern machine translation systems, they are reasonable for the task of hieroglyphic transliteration. An average Levenshtein distance of around 0.10 per character roughly means one incorrect character out of ten, showing that most predictions are very close to the reference transliterations. Considering the limited amount of training data and the specific properties of hieroglyphic writing, these results can be seen as a good outcome for this task.

Model	**Word Accuracy (%)**	**Character Accuracy (%)**	**Levenshtein / char.**	**ROUGE-L**	**SacreBLEU**	**Gold**
PreviousBest	**78.55**	61.21	0.1010	89.00	84.14	**1292**
OptunaBest	78.01	**61.50**	**0.0993**	**89.12**	**84.65**	1278

Table 4.5: Comparison of the previous best model ($h = 4$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$) and the best model found by automatic search ($h = 4$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$).

Both the manual tuning approach used in previous work and the automatic tuning approach based on Optuna achieved comparable results. However, the effort required to obtain these results differs substantially. Manual tuning required approximately one month of continuous trial-and-error experimentation, including the selection of promising hyperparameter combinations and full training of multiple models. This process is both time-consuming and computationally expensive.

In contrast, the automated hyperparameter search is considerably more efficient. Although the exact duration depends on the optimizer configuration and caching behavior, the overall search is estimated to require less than 24 hours, followed by approximately 55 hours for training of the final selected model, and approximately 1 hour of testing. This approach is therefore significantly more efficient in terms of both time and computational resources, as only a single model needs to be fully trained.

To compare these models in terms of size, the previously found model has 2.17 million trainable parameters, while the model found by the automated hyperparameter search has 3.22 million parameters, meaning the newly found model is slightly larger.

Prediction Comparison As discussed in [Landgráfová et al. 2025], transformer-based transliteration models trained on the Ramses corpus reveal systematic divergences that reflect both genuine linguistic ambiguity and occasional errors in human-annotated datasets.

Out of all 2,729 testing lines, 1,119 (41.00%) were correctly predicted by both models, 417 (15.28%) were incorrectly predicted by both models with the same mistake, and 891 (32.65%) were incorrectly predicted by both models with different mistakes.

Example 1: Both Models Make the Same Mistake (Line 7 of the testing file)

Source:

```
O42 Q3 A24 _ Z7 X1 V31 _ G41 G1 N5 G7 _ N35 _ O42 Q3 A24 _ F35 _
```

Target:

```
S s p _ t w k _ p A - r a _ m _ S s p _ n f r _
```

Both Predictions:

```
S s p _ t w k _ p A - r a _ n _ S s p _ n f r _
```

This type of mistake, where both models incorrectly predict 'n' instead of 'm', reflects the orthographic ambiguity inherent in Ancient Egyptian transliteration. As noted in [Landgráfová et al. 2025], in Late Egyptian, 'm' and 'n' are often interchangeable.

Example 2: Both Models Make the Same Mistake (Line 20 of the testing file)

Source:

```
M17 Z7 _ D37 _ A1 _ T18 S29 Z7 D54 _ I9 _
```

Target:

```
i w _ r d i _ = i _ S m s _ = f _
```

Both Predictions:

```
i w _ r d i _ = i _ S m s w _ = f _
```

Among the 417 cases where both models make the same mistake, the most frequent errors involve the 'w' token: missing 'w' tokens occur 78 times (18.7% of shared mistakes), while extra 'w' tokens appear 51 times (12.2% of shared mistakes). Additionally, confusion between 'n' and 'm' tokens appears 12 times (2.9% of shared mistakes), where both models incorrectly predict 'n' instead of the target 'm'.

These 417 shared mistakes may indicate potential errors or ambiguities in the annotated dataset itself. Identifying cases where two independently trained models make the same incorrect prediction can be particularly valuable for Egyptologists, as it highlights problematic parts of the corpus that may require closer inspection or reannotation. Such cases allow egyptologists to apply their specialized knowledge to reassess uncertain transliterations.

In contrast, the 891 cases where both models are incorrect but produce different predictions are more likely to reflect genuinely difficult or ambiguous phrases. These examples help reveal the limitations of the models and provide insight into challenging linguistic patterns.

4.2.3.6 Saving Computational Time: Cross-Study Caching

This analysis evaluates the effectiveness of the cross-study caching mechanism described in Section 3.3. The caching system was designed to avoid repeated evaluations of identical hyperparameter configurations across multiple optimization studies when comparing different optimizers (TPE, GP, Random, and Grid).

Different optimization algorithms often explore overlapping regions of the hyperparameter space. Without caching, this leads to repeated evaluations of the same configurations, resulting in unnecessary computational cost—especially when running several studies with different optimizers for comparison.

For each optimizer, the actual runtime of individual trials was extracted where available. However, not all timing information was recorded, particularly during the initial experiments. As a result, training times are missing for 22 trials. To estimate the total time saved by the caching mechanism, the average trial duration was used for trials with missing timing information. Consequently, the reported time savings should be considered approximate.

Table 4.6 summarizes the cache usage and estimated time savings for each optimizer. On average across all studies, a single trial took approximately 0.253 hours (about 15.18 minutes). For each optimizer, the time saved was calculated based on the number of configurations retrieved from the cache and their associated training times. When a cached trial did not have a recorded duration, the average trial time was used instead.

Optimizer	**Total Trials**	**Cached Trials**	**Cache Usage**	**Actual Time (h)**	**Time Saved (h)**
TPE	50	28	56.0%	5.60	4.98
GP	50	25	50.0%	5.73	5.48
Random	50	28	56.0%	5.87	6.94
Grid	50	46	92.0%	1.28	9.14
Total	**200**	**127**	**63.5%**	**18.47**	**26.54**

Table 4.6: Approximate cache performance by optimizer.

Across all 200 trials, the caching mechanism achieved a cache hit rate of 63.5%, resulting in substantial computational savings. The total actual runtime of all studies was 18.47 hours, while the estimated runtime without caching would have been approximately 45.01 hours. Overall, these results demonstrate that cross-study caching can significantly improve computational efficiency in large-scale hyperparameter optimization experiments.

4.2.4 Model Size vs Model Performance Comparison

Previous experiments (such as [Morovicsová 2024]) showed that in some cases, if the model is too large compared with the amount of data available, it has limited potential to learn effectively. I investigated this potential trend by plotting the dependency between the maximum validation accuracy achieved by the models and the number of trainable parameters. The results can be seen in Figure 4.6.

The models present in this comparison were trained on 30% of the Ramses dataset and were trained as part of the Main experiment described in Section 4.2.3.

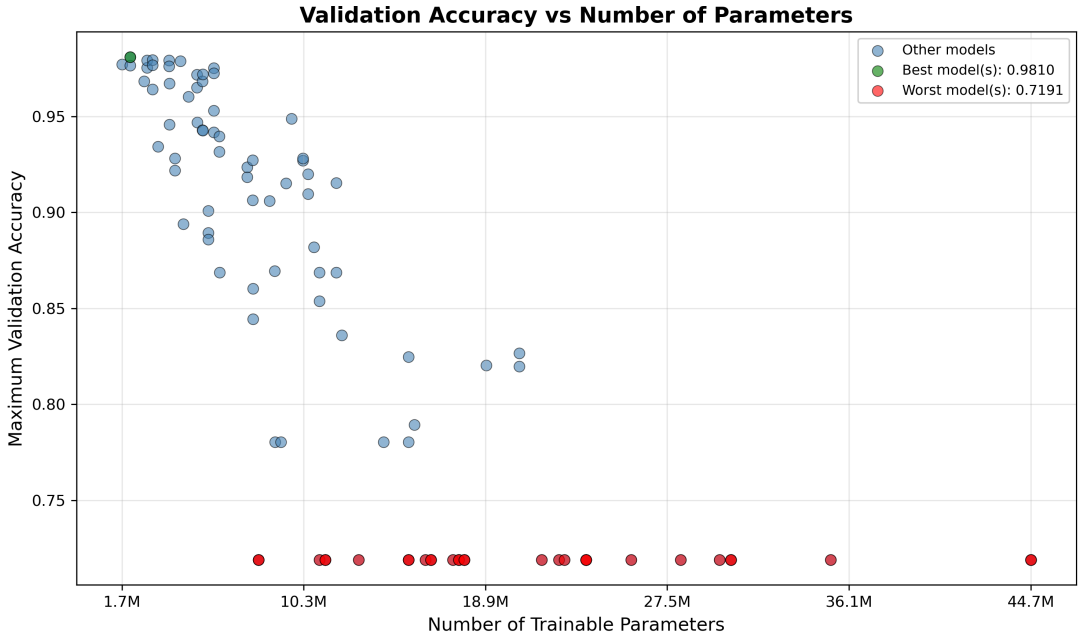
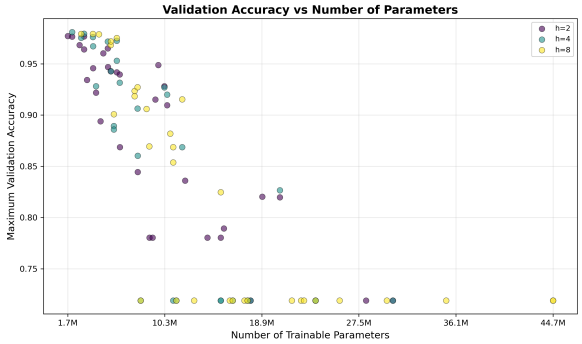


Figure 4.6: Validation accuracy vs number of parameters.

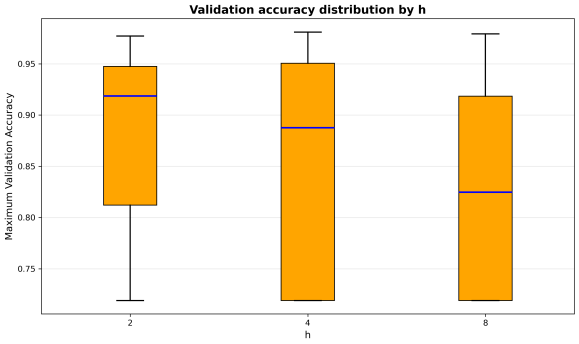
Furthermore, different hyperparameter settings significantly influence the performance of the model. Figures 4.7, 4.8, 4.9, 4.10, and 4.11 show how individual hyperparameters affect validation accuracy. Each figure presents both a scatterplot colored by the respective hyperparameter and a boxplot showing the distribution of validation accuracy across different values of that hyperparameter.

The model with the fewest parameters achieved a validation accuracy of 97.70% with only 1.7 million parameters using hyperparameters $h = 2$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, and $n = 2$. Remarkably, the best-performing model reached a validation accuracy of 98.10% with just 2.1 million parameters and hyperparameters $h = 4$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, and $n = 2$ - only slightly more parameters than the smallest model. In contrast, the largest model with 44.7 million parameters and hyperparameters $h = 8$, $d_k = 64$, $d_{ff} = 2048$, $d_{model} = 512$, and $n = 6$, achieved (similarly to several other models) the lowest validation accuracy of 71.91%. This reveals a pattern that goes against the assumption that larger models perform better. It demonstrates that in translation problems with limited data, increasing the number of parameters does not guarantee better performance; in fact, the largest models in these experiments performed significantly worse than smaller models.

Upon further analysis, the models that performed worst were trained for the fewest epochs - the training in these cases was stopped by the Early Stopping Callback, since the validation accuracy was either not improving or even dropping significantly. This suggests that the models were either unable to learn from the available data or overfitting on the training data occurred, which caused the validation accuracy to drop.

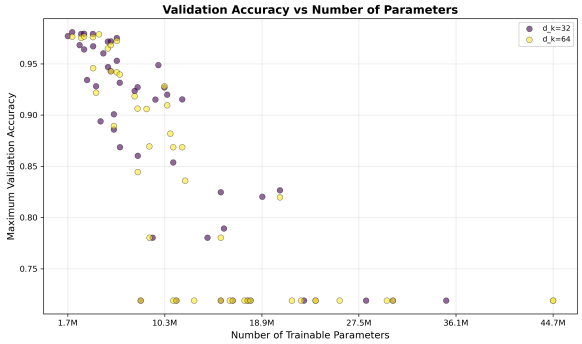


(a) Colored by h

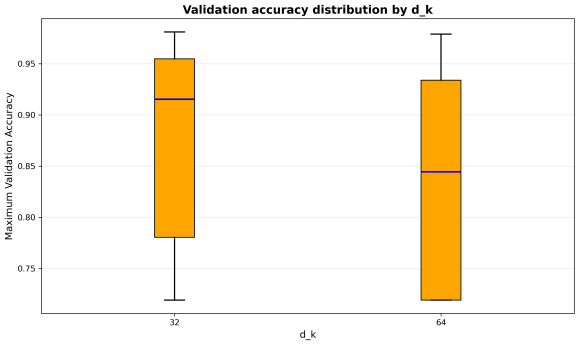


(b) Boxplot for h

Figure 4.7: Validation accuracy dependency on hyperparameter h.

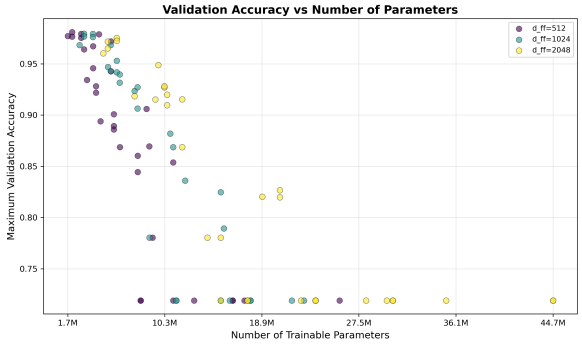


(a) Colored by d_k

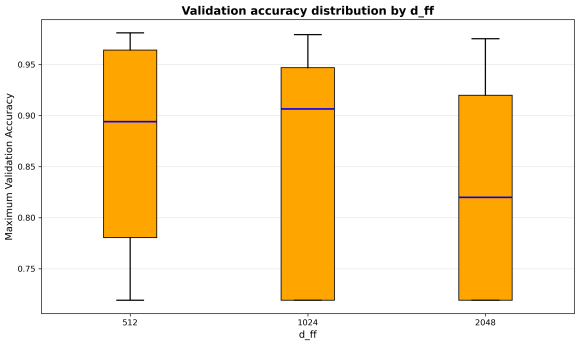


(b) Boxplot for d_k

Figure 4.8: Validation accuracy dependency on hyperparameter d_k.

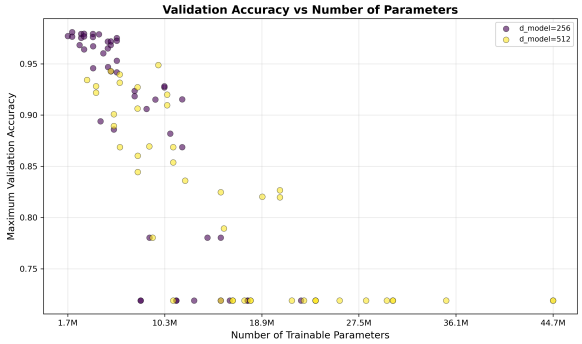


(a) Colored by d_{ff}

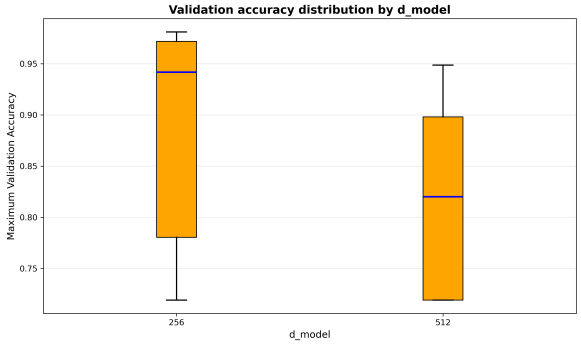


(b) Boxplot for d_{ff}

Figure 4.9: Validation accuracy dependency on hyperparameter d_{ff}.

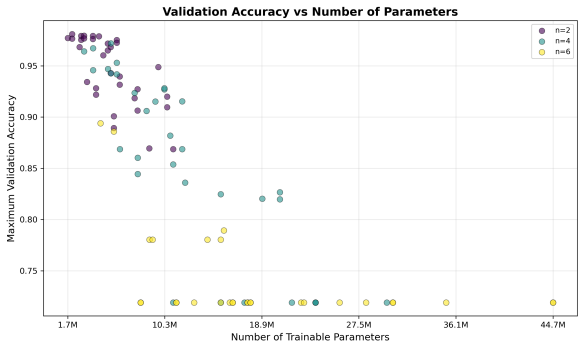


(a) Colored by d_{model}

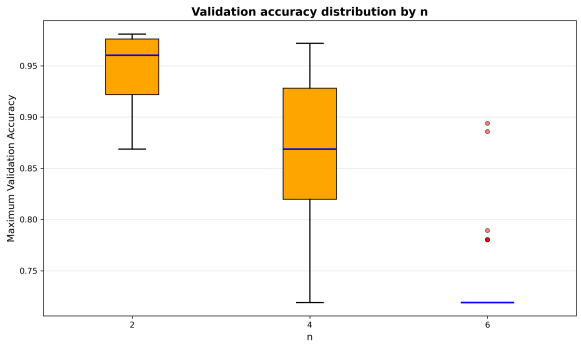


(b) Boxplot for d_{model}

Figure 4.10: Validation accuracy dependency on hyperparameter d_{model}.



(a) Colored by n



(b) Boxplot for n

Figure 4.11: Validation accuracy dependency on hyperparameter n.

4.2.5 Comparing Stochastic Optimizers

To evaluate the reproducibility and consistency of different stochastic hyperparameter optimization strategies, I conducted a study comparing GP and TPE. Similarly to the main experiment (Section 4.2.3), all evaluations were performed on a 30% subset of the original Ramses dataset.

For each optimizer, I ran 100 independent experiments with different random seeds. With each seed, the optimizer was given a maximum of 20 trials to try to find the optimal configuration. Thanks to the cross-study caching mechanism, this required low computational resources. This setup allows for analyzing the average performance, stability, convergence behavior, and sensitivity to random initialization.

4.2.5.1 Distribution of Best Accuracies

Figure 4.12 shows the distribution of best validation accuracies (the highest validation accuracy achieved in that trial) across all 100 runs for GP and TPE. GP achieved a higher mean and lower variance, indicating more stable convergence behavior.

- **GP**: Mean = 0.9808, Std = 0.0005
- **TPE**: Mean = 0.9777, Std = 0.0047

GP consistently converges to accuracies close to 0.9810, while TPE shows a wider spread of results. This suggests that TPE is generally more explorative than GP, which appears to focus more on exploitation.

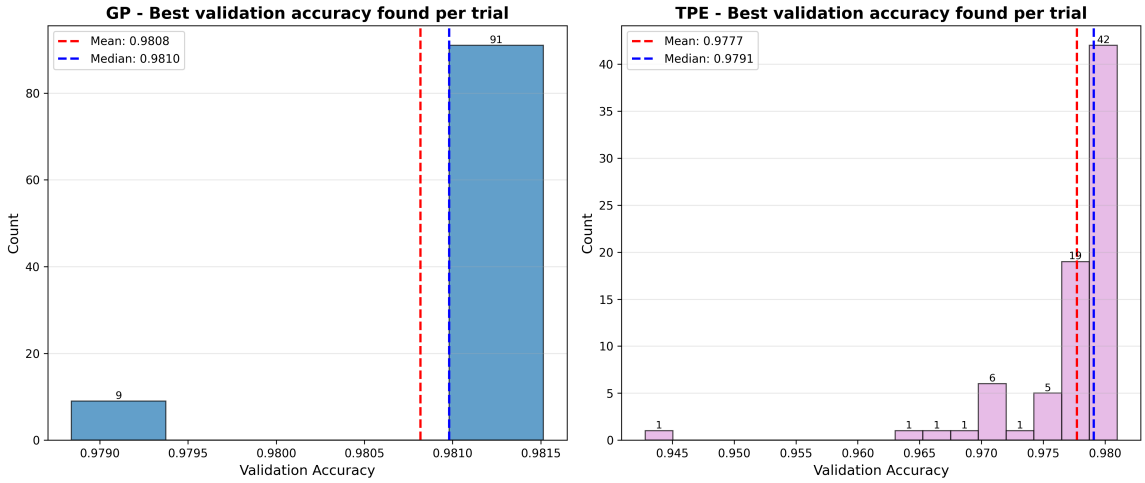


Figure 4.12: Distribution of best validation accuracies across 100 seeds for GP and TPE.

A boxplot comparison in Figure 4.13 summarizes the two optimizers' performance distributions. GP exhibits a tight cluster around 0.981, while TPE shows a wider spread and several low-performing outliers, indicating its more explorative nature. GP demonstrates both higher accuracy and lower variance compared to TPE.

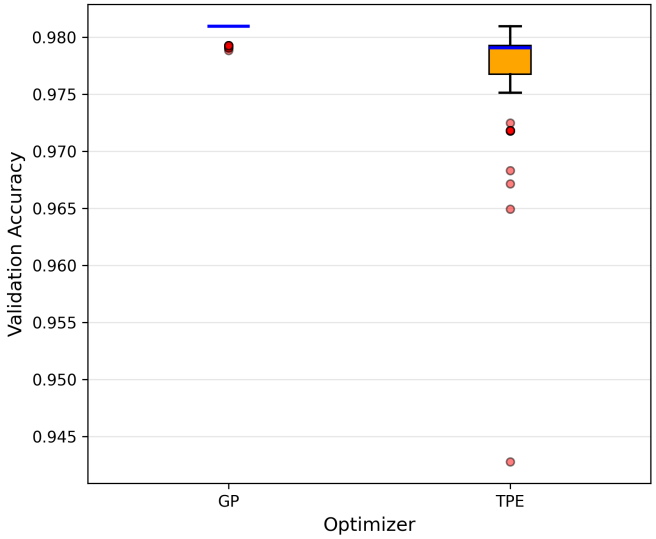


Figure 4.13: Boxplot comparison of best validation accuracies.

4.2.5.2 Convergence Speed Analysis

Both methods show rapid initial gains within the first few trials. The differences are that GP curves cluster tightly, with most runs reaching the best validation accuracy around the same trial while TPE curves exhibit higher variability, with some runs converging early and others requiring the full 20 trials.

I further examined the trial number at which each optimizer first reached its best validation accuracy, the results are in Figure 4.14. TPE converges earlier on average, but GP achieves higher final accuracy.

- **GP:** Mean convergence at 14.1 trials
- **TPE:** Mean convergence at 11.0 trials

Although overall GP achieves higher accuracy than TPE, it usually requires more trials to reach it, while TPE tends to peak in earlier trials.

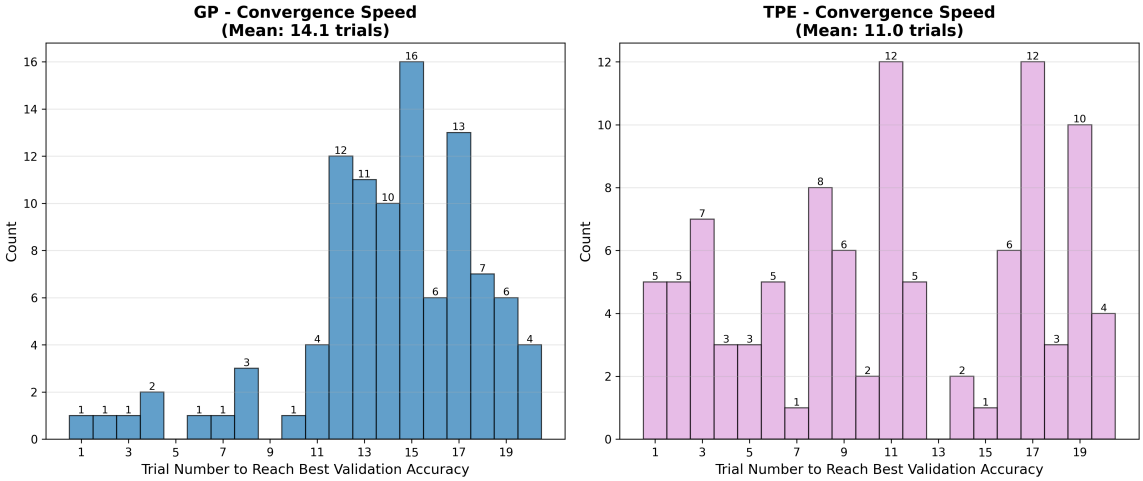


Figure 4.14: Histogram of convergence trial index for GP and TPE.

4.2.5.3 Optimizer Speed Comparison

To evaluate the computational overhead of different optimization samplers, I measured the time required for parameter suggestion across 100 trials. The results demonstrate a significant performance difference between samplers: the Tree-structured Parzen Estimator (TPE) sampler averaged 8.96 ms per suggestion with a median of 9.95 ms, while the Gaussian Process (GP) sampler averaged 301.22 ms per suggestion with a median of 295.63 ms, making TPE approximately 33.6 times faster than GP on average. Additionally, GP exhibited substantially higher variance in suggestion times (ranging from 0.50 ms to 1703.44 ms) compared to TPE (ranging from 0.42 ms to 15.05 ms). However, it is important to note that the time needed for parameter suggestion is extremely short compared to the actual training time of neural networks.

4.2.5.4 Summary

Overall, the experiment demonstrates that GP-based optimization achieves higher average performance and lower variance, showing better robustness across different random seeds. TPE, on the other hand, is more explorative. Both approaches are valid choices for selecting the best model for hieroglyphic transliteration, although the results suggest that GP consistently finds the most accurate model configuration on average across multiple runs and may therefore be a more suitable choice than TPE.

4.3 Results: Dataset Size Dependency Study

To summarize the experiment (described in detail in Section 3.4): This experiment aims to quantify how dataset size affects neural translation quality in a low-resource setting. In this experiment, the full Ramses corpus was divided into 67 progressively larger subsets, starting with 1,000 sentences and ending with the entire dataset. A separate model was trained on each subset and then evaluated on the same test portion. The results here show how performance shifted across these scaled datasets and how the model responded to each increase in training size.

I first conducted the experiment using a small model to estimate approximately where the performance plateau might occur. Based on these findings, I then trained and evaluated selected larger models on specific dataset sizes.

4.3.1 Initial experiment

This initial experiment was conducted using a small Transformer model to indicate how performance changes with increasing amounts of training data and also to verify that the full training pipeline operates correctly before training larger models. The model used in this phase had the following configuration:

$$\text{Model: } h = 2,\ d_k = 32,\ d_{ff} = 128,\ d_{model} = 32,\ n = 1$$

This model is very small, having around 83k trainable parameters.

Each subset of the Ramses corpus was used to train a fresh instance of this model for up to 20 epochs, and both validation accuracy and validation loss were recorded for every run.

As shown in Figure 4.15, the resulting curves reveal a clear pattern. Validation accuracy increases rapidly at the smallest dataset sizes, particularly between 1,000 and 10,000 sentences, while the validation loss decreases sharply over the same interval. This demonstrates that the model immediately benefits from additional data when starting from very limited training resources.

As the subsets grow larger, the improvements gradually slow: between roughly 20,000 and 30,000 sentences, accuracy still rises but only in small increments, and the loss stabilizes with minor fluctuations. Once the dataset reaches approximately 30,000–35,000 sentences, both metrics enter a visible plateau. This behavior aligns with the plateau hypothesis, reflecting strong gains when data is scarce followed by diminishing returns as the model approaches the limits of what it can learn given its capacity.

4.3.2 Main experiment

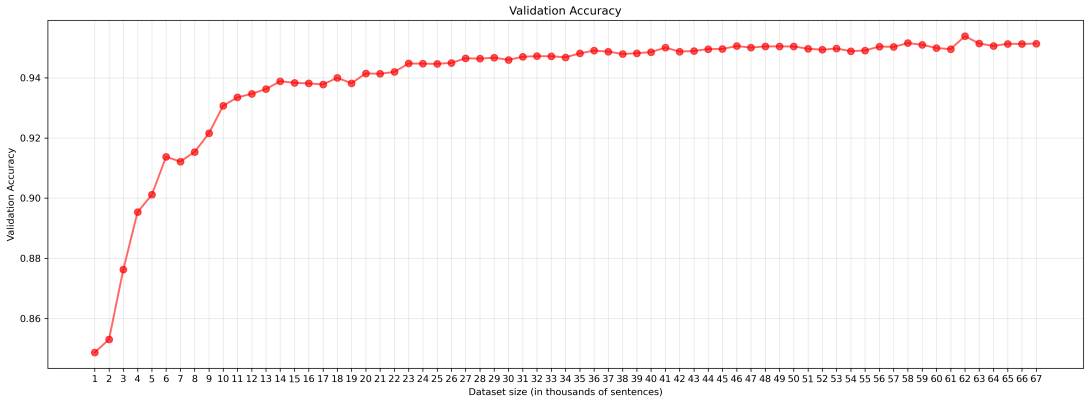
In the main experiment, I decided to compare 3 models on selected subcorpus sizes, as training models on every subcorpus size in the plateau region is not necessary. I selected the following models:

- Small model: $h = 2$, $d_k = 32$, $d_{ff} = 128$, $d_{model} = 32$, $n = 1$, $params = 82911$
- Large model: $h = 2$, $d_k = 64$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$, $params = 2.1\,\text{M}$
- Model 3: $h = 4$, $d_k = 64$, $d_{ff} = 512$, $d_{model} = 512$, $n = 2$, $params = 5.9\,\text{M}$

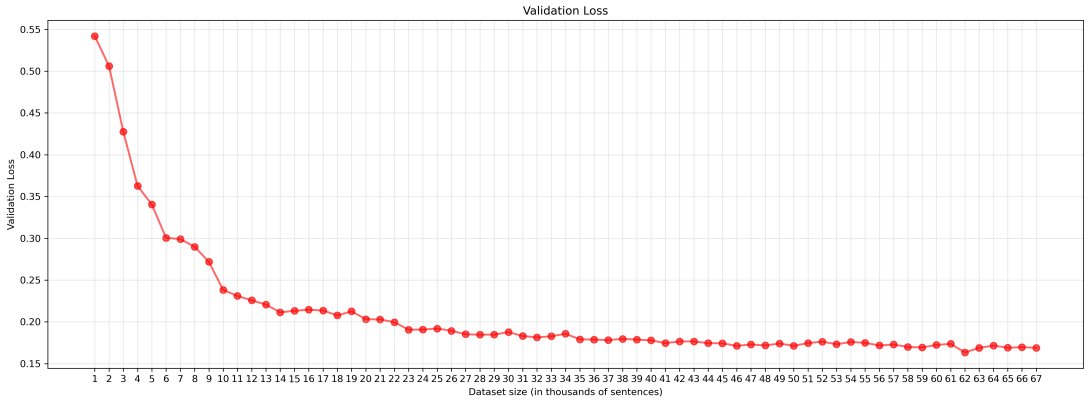
The Small model has the same configuration as the small model tested in the initial experiment. The Large model corresponds to the best model found in my Bachelor's thesis [Morovicsová 2024]. Model 3 is a comparatively large model which I have also previously trained as part of my Bachelor's thesis.

To reduce resource requirements of this experiment, the above mentioned models were trained on subcorpora of the following sizes (in thousands of sentences):

$$1, 2, 3, 4, 5, 6, 7, 8, 9, 13, 19, 25, 31, 37, 43, 49, 55, 61, 67$$



(a) Validation accuracy for small model



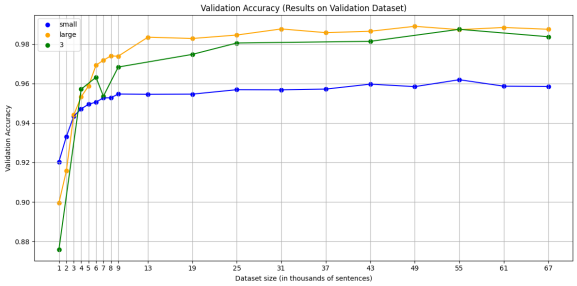
(b) Validation loss for small model

Figure 4.15: Model performance as a function of dataset size. (a) Validation accuracy improves from approximately 0.85 at 1,000 samples to around 0.95 at 67,000 samples. (b) Validation loss decreases consistently with increasing dataset size, reaching values around 0.17 for the largest datasets.

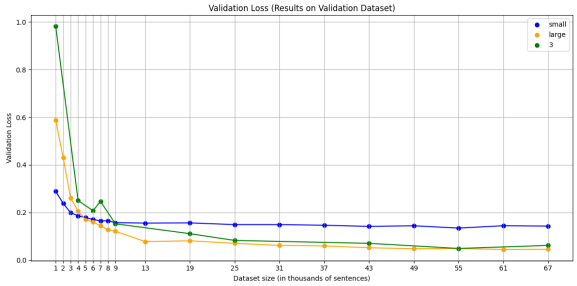
Figure 4.16 shows the validation accuracy and loss metrics obtained during training. The results in Figure 4.17 show a similar plateau pattern for Levenshtein distance on the testing dataset.

The Small model improves quickly with more data up to around 10,000 sentences but then levels off, which is consistent with its limited size and capacity. The Large model, with more parameters, keeps improving steadily over a wider range of dataset sizes and outperforms the Small model noticeably in the middle data range before it starts to plateau. This means larger models can make better use of more data before reaching their limits. In contrast, Model 3, the largest model, does not outperform the Large model and sometimes performs worse, especially with lower amounts of data. This suggests that an overly large model may not be well suited for low-resource tasks.

Overall, the results highlight how model size and data amount work together: small models reach their maximum performance quickly, while larger ones need more data but eventually face similar diminishing returns. These results indicate that, even for larger models, the training process still follows the plateau pattern observed in the initial experiment.



(a) Validation accuracy vs. dataset size



(b) Validation loss vs. dataset size

Figure 4.16: Comparison of validation performance across different dataset sizes.

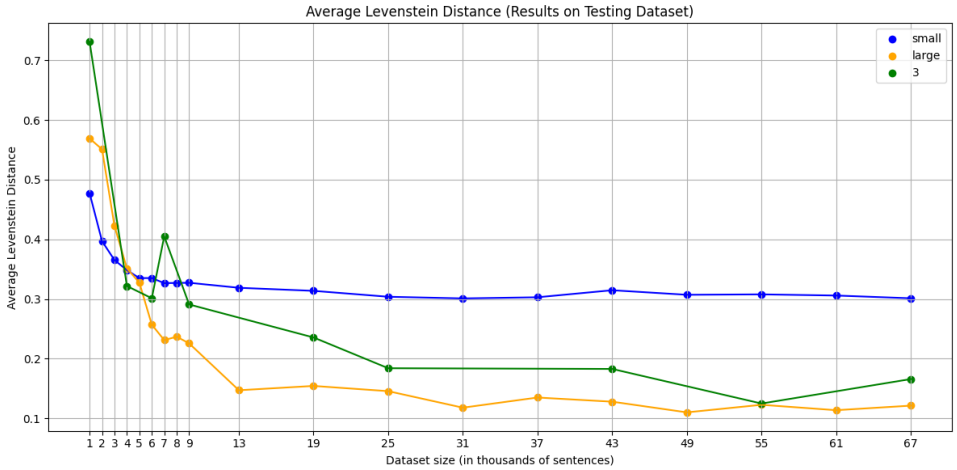


Figure 4.17: Comparison of average Levenshtein distance per valid character across different dataset sizes.

4.3.2.1 Summary

These findings suggest that the minimum size of the dataset required to successfully train a transformer-based model on hieroglyphic data is somewhere between 10 and 25 thousand sentences, which corresponds to the regions where the selected models start to plateau.

4.3.3 Dependency on Initial Conditions

To further assess the performance of each model, I did an experiment to evaluate the sensitivity to initialization. The best-performing model architecture (referred to as the Large model in previous text) was selected and trained on three subsets of the original Ramses dataset, containing respectively 7, 9, and 13 thousand sentences. For each subset, the model was trained 10 times using 10 different random seeds, resulting in 10 distinct initial conditions. Each training run was carried out for 200 epochs. The resulting models were then evaluated on the test dataset.

The results, shown in Figure 4.18, indicate variability in the average Levenshtein distance computed on the test set across different initializations. This variability suggests that the model's performance is highly sensitive to the choice of initial conditions. Certain initializations lead to significantly worse performance, which implies that additional training or alternative initialization strategies may be needed to achieve optimal results.

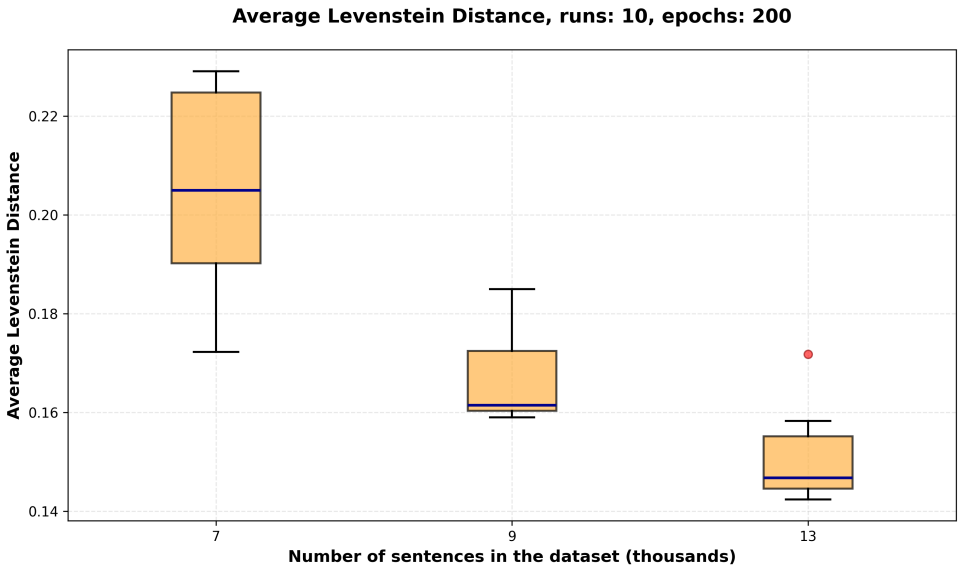


Figure 4.18: Comparison of the effects of different initializations. The best-performing model (Large model: $h = 2$, $d_k = 64$, $d_{ff} = 512$, $d_{model} = 256$, $n = 2$, 2.1 M parameters) was trained 10 times on each of three subsets of the training data containing 7, 9, and 13 thousand sentences.

Chapter 5

Discussion

This work presents a comprehensive evaluation of hyperparameter optimization methods for Transformer-based models applied to the task of Ancient Egyptian hieroglyph transliteration. The experimental results demonstrate that systematic hyperparameter optimization can achieve performance comparable to manual tuning while requiring significantly less time and effort. The best model found through automated search achieved a validation accuracy of 98.10% with hyperparameters $h = 4$, $d_k = 32$, $d_{ff} = 512$, $d_{model} = 256$, and $n = 2$, performing comparably to the manually tuned model from previous work.

The comparison of optimization methods confirms that Bayesian optimization approaches (TPE and Gaussian Processes) are more efficient than non-Bayesian methods. Both TPE and GP achieved the best validation accuracy of 0.9810 within just 8 trials, while Grid Search required 42 trials to reach a slightly lower accuracy of 0.9793. The stochastic optimizer comparison study reveals that Gaussian Processes demonstrate superior stability, achieving a mean validation accuracy of 0.9808 across 100 independent runs, compared to TPE's mean of 0.9777.

Another finding is the relationship between model size and performance in low-resource settings. The best-performing model achieved 98.10% validation accuracy with only 2.1 million parameters, while the largest model (44.7 million parameters) achieved the lowest accuracy of 71.91%. This suggests that the Transformer architecture may be approaching the limits of what can be learned from the available hieroglyphic data, and that simply increasing model capacity is not a viable path to improvement.

The dataset size dependency study confirms the plateau hypothesis, showing that model performance improves rapidly with increasing data and then reaches a plateau. This finding is consistent across different model sizes, suggesting that the plateau is a fundamental property of the task. However, the sensitivity to initialization observed in the experiments suggests that additional training strategies may be necessary to achieve more consistent results.

Several limitations of this work should be acknowledged. The hyperparameter search space was deliberately constrained to enable fair comparison between optimizers, meaning that potentially optimal configurations outside the predefined search space were not explored. The experiments were conducted on a single dataset. Additionally, the Transformer architecture itself may represent a limitation, as the observation that larger Transformers perform worse suggests that the architecture may be approaching its limits for this task.

Several directions for future research emerge from this work. Expanding the hyperparameter search to a continuous or significantly larger discrete space using Bayesian optimizers could potentially discover better configurations. Another possible direction is the use of active learning techniques, which aim to reduce labeling or evaluation costs by selecting informative queries. Incorporating additional datasets would provide more training data, though this would require addressing challenges in data unification and reconciling different transliteration systems used across the field. Investigating alternative model

architectures or training strategies, such as specialized architectures for low-resource settings or transfer learning, could address the limitations observed with large Transformers. Finally, research into improved initialization strategies or training techniques could yield more consistent results given the sensitivity to initialization observed in the experiments.

Chapter 6

Conclusion

This work presents a systematic evaluation of hyperparameter optimization methods for Transformer-based models in the context of Ancient Egyptian hieroglyph transliteration, a low-resource machine translation task. The primary contribution is the demonstration that automated hyperparameter optimization can achieve performance comparable to manual tuning while requiring significantly less time and computational resources. The best model discovered through systematic search achieved a validation accuracy of 98.10% and performed comparably to manually tuned models on test metrics, including character-level accuracy (61.50%), average Levenshtein distance per character (0.0993), ROUGE-L (89.12), and SacreBLEU (84.65). This result was obtained with less than 24 hours of automated search followed by 55 hours of final model training, compared to approximately one month of manual trial-and-error experimentation required in previous work.

The comparative study confirms that Bayesian optimization approaches (TPE and Gaussian Processes) are more efficient than non-Bayesian methods for this task. Both TPE and GP achieved optimal performance within 8 trials, while Grid Search required 42 trials. More importantly, Gaussian Processes provide superior stability across different random initializations, achieving a mean validation accuracy of 0.9808, compared to TPE's mean of 0.9777.

A key finding is the relationship between model size and performance in low-resource settings. The best-performing model achieved 98.10% validation accuracy with only 2.1 million parameters, while the largest model (44.7 million parameters) achieved the lowest accuracy of 71.91%. This highlights the importance of matching model capacity to available data and suggests that the Transformer architecture may be approaching its limits for this specific task given the current dataset size.

The dataset size dependency study identifies a performance plateau, beyond which additional training data provides diminishing returns. The work also demonstrates the practical value of cross-study caching in hyperparameter optimization, achieving a 63.5% cache hit rate and saving approximately 26.54 hours of computation time.

In summary, this work establishes that systematic hyperparameter optimization is highly beneficial for low-resource machine translation tasks. The results indicate that, for hieroglyphic transliteration, moderately sized or smaller Transformer models optimized using Bayesian methods are the most suitable choice.

Bibliography

Akiba, Takuya et al. [2019]. "Optuna: A Next-generation Hyperparameter Optimization Framework." In: *CoRR* abs/1907.10902. URL: http://arxiv.org/abs/1907.10902.

Allen, James P. [2005]. *The Ancient Egyptian Pyramid Texts*. Vol. 23. Writings from the Ancient World. Brill. ISBN: 978-90-04-13777-6. URL: https://brill.com/edcollbook/title/11082?language=en.

Ardagh, Philip [1999]. *The Hieroglyphs Handbook: Teach Yourself Ancient Egyptian*. Faber & Faber. ISBN: 9780571197446.

Bergstra, James, Rémi Bardenet, et al. [2011]. "Algorithms for Hyper-Parameter Optimization." In: *Advances in Neural Information Processing Systems*. Vol. 24, pp. 2546–2554. URL: https://proceedings.neurips.cc/paper/2011/hash/86e8f7ab32cfd12577bc2619bc635690-Abstract.html.

Bergstra, James and Yoshua Bengio [2012]. "Random Search for Hyper-Parameter Optimization." In: *Journal of Machine Learning Research* 13, pp. 281–305.

Bergstra, James, Daniel Yamins, and David Cox [2013]. "Making a Science of Model Search: Hyperparameter Optimization in Hundreds of Dimensions for Vision Architectures." In: *Proceedings of the 30th International Conference on Machine Learning*.

Brochu, Eric, Vlad M. Cora, and Nando de Freitas [2010]. "A Tutorial on Bayesian Optimization of Expensive Cost Functions, with Application to Active User Modeling and Hierarchical Reinforcement Learning." In: *arXiv preprint arXiv:1012.2599*.

Cohn, David A., Les Atlas, and Richard E. Ladner [1994]. "Improving generalization with active learning." In: *Machine Learning* 15.2, pp. 201–221. DOI: 10.1023/A:1022673506211.

De Cao, Mattia et al. [2024]. "Deep Learning Meets Egyptology: a Hieroglyphic Transformer for Translating Ancient Egyptian." In: *Proceedings of the 1st Workshop on Machine Learning for Ancient Languages (ML4AL 2024)*. Association for Computational Linguistics. URL: https://aclanthology.org/2024.ml4al-1.9/.

Gardiner, Alan H. [1957]. *Egyptian Grammar: Being an Introduction to the Study of Hieroglyphs*. Oxford.

Haddow, Barry et al. [2022]. "Survey of Low-Resource Machine Translation." In: *Computational Linguistics* 48.3, pp. 673–732. DOI: 10.1162/coli_a_00446. URL: https://aclanthology.org/2022.cl-3.6/.

Hsu, Chih-Wei, Chih-Chung Chang, and Chih-Jen Lin [2003]. *A practical guide to support vector classification*. Tech. rep. Department of Computer Science, National Taiwan University.

Hutter, Frank, Holger Hoos, and Kevin Leyton-Brown [22–24 Jun 2014]. "An Efficient Approach for Assessing Hyperparameter Importance." In: *Proceedings of the 31st International Conference on Machine Learning*. Ed. by Eric P. Xing and Tony Jebara. Vol. 32. Proceedings of Machine Learning Research 1. Bejing, China: PMLR, pp. 754–762. URL: https://proceedings.mlr.press/v32/hutter14.html.

Jauhiainen, Heidi and Tommi Jauhiainen [2023]. "Transliteration Model for Egyptian Words." In: *Digital Humanities in the Nordic and Baltic Countries Publications* 5.1, pp. 149–164. DOI: 10.5617/dhnbpub.10659.

Landgráfová, Renata et al. [2025]. "Towards AI-assisted translation of hieroglyphic Egyptian: interpreting Iufaa's Coffin Texts." In: *Zeitschrift für Ägyptische Sprache und Altertumskunde*. submitted, manuscript ID: ZAES.2025.0029.

Lin, Chin-Yew [2004]. "ROUGE: A Package for Automatic Evaluation of Summaries." In: *Text Summarization Branches Out*. Barcelona, Spain: Association for Computational Linguistics, pp. 74–81. URL: https://aclanthology.org/W04-1013/.

Morovicsová, Katka [2024]. *Sequence translations and their applications*. Bachelor's thesis. Prague, Czech Republic. URL: http://hdl.handle.net/10467/117302.

Papineni, Kishore et al. [2002]. "Bleu: a Method for Automatic Evaluation of Machine Translation." In: *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*. Philadelphia, Pennsylvania, USA: Association for Computational Linguistics, pp. 311–318. DOI: 10.3115/1073083.1073135. URL: https://aclanthology.org/P02-1040/.

Post, Matt [2018]. "A Call for Clarity in Reporting BLEU Scores." In: *Proceedings of the Third Conference on Machine Translation: Research Papers*. Brussels, Belgium: Association for Computational Linguistics, pp. 186–191. URL: https://aclanthology.org/W18-6319/.

Rasmussen, Carl Edward and Christopher K. I. Williams [2006]. *Gaussian Processes for Machine Learning*. MIT Press. ISBN: 026218253X.

Rosmorduc, Serge [Dec. 2020]. "Automated Transliteration of Late Egyptian Using Neural Networks." In: *Lingua Aegyptia - Journal of Egyptian Language Studies*. Lingua Aegyptia 28, pp. 233–257. DOI: 10.37011/lingaeg.28.07. URL: https://hal.science/hal-03118369.

Shaw, Ian [2003]. *The Oxford History of Ancient Egypt*. 2nd. Oxford, UK: Oxford University Press.

Snoek, Jasper, Hugo Larochelle, and Ryan P Adams [2012]. "Practical Bayesian Optimization of Machine Learning Algorithms." In: *Advances in Neural Information Processing Systems*. Vol. 25, pp. 2951–2959.

Vaswani, Ashish et al. [2017]. *Attention Is All You Need*. https://arxiv.org/abs/1706.03762. arXiv: 1706.03762 [cs.CL].

Washington, University of [2003]. *Hieroglyphic E-Convert Application (HECA)*. https://depts.washington.edu/llc/external/hecav/index-1.3.htm. Dictionary based on Sir Alan Gardiner's Egyptian Grammar.

Watanabe, Shuhei [2023]. "Tree-Structured Parzen Estimator: Understanding Its Algorithm Components and Their Roles for Better Empirical Performance." In: *arXiv preprint arXiv:2304.11127*. DOI: 10.48550/arXiv.2304.11127.